Haskell Avanzado: Mónadas – I/O

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20

Mónadas

Mónadas

Una idea fundamental de los lenguajes funcionales es la idea de las funciones **puras**:

- toman un valor y devuelven un resultado
- para el mismo valor el resultado siempre es el mismo
- no tienen efectos laterales

Las funciones puras

- son más fáciles de verificar
- son predecibles
- son más fáciles de depurar
- permiten generar mejor código

Sin embargo, existen situaciones en las que el código con efectos laterales podría ser más sencillo de escribir, o es imposible de implementar con funciones puras.

Ideas...

Piensa varios ejemplos que NO puedan hacerse utilizando funciones puras

Situaciones en las que las funciones puras no sirven (además de las que habéis pensado antes):

- Interacción con el usuario.
- Lectura y escritura por pantalla.
- Situaciones en las que es obligatorio guardarse un *estado* (por ejemplo, números pseudo-aleatorios).
- *Debugging*

Todos los lenguajes funcionales (Haskell incluido) deben proporcionar soluciones a estos problemas de los lenguajes funcionales puros.

La pregunta más difícil de la asignatura (formulada de diferentes formas):

- ¿Cómo añadimos efectos laterales a una función sin efectos laterales?
- ¿Cómo permitimos que una función cambie un estado de ejecución?
- ¿Cómo conseguir mediante funciones la ejecución de varias acciones secuencialmente?

- ¿Cómo añadimos efectos laterales a una función sin efectos laterales?

No podemos añadir efectos laterales. Sólo podemos hacer que devuelva (o tenga como parámetros) datos más complicados que se vayan concatenando.

- ¿Cómo permitimos que una función cambie un estado de ejecución?

No podemos cambiar el estado de ejecución. Sin embargo, una función puede recibir un parámetro que represente dicho estado y transformarlo para devolverlo.

- ¿Cómo conseguir mediante funciones la ejecución de varias acciones secuencialmente?

No puedes .[ej]ecutar funciones secunecialmente". Sin embargo se tiene cierto control del orden de ejecución mediante la **composición de funciones**.

¿Cómo hacemos para *debuggear* estas funciones? (*debuggear* $\rightarrow$ lanzar un mensaje de texto)

```
1 f, g :: Float -> Float
```

¿Cómo hacemos para *debuggear* estas funciones? (*debuggear* $\rightarrow$ lanzar un mensaje de texto)

```
f, g :: Float -> Float
```

Definiendo funciones alternativas

```
f', g' :: Float -> (Float, String)
```

donde la cadena de texto tiene el mensaje correspondiente.

```
addDebug :: Float -> (Float, String)
addDebug x = (x+1, " (Add one) ")
mulDebug :: Float -> (Float, String)
mulDebug x = (x*2, " (Mul two) ")
```

```haskell
f, g :: Float -> Float
f', g' :: Float -> (Float, String)
```

Problema: El nuevo parámetro provoca que las nuevas funciones no encajen como las anteriores. Por ejemplo, podemos hacer

```haskell
f . g
```

pero no podemos hacer

```haskell
f' . g'
```

porque los tipos de datos han cambiado y no encajan.

```haskell
f', g' :: Float -> (Float, String)
```

Nueva función para que encajen (concatenando mensajes de error):

```haskell
bind :: (Float -> (Float,String))
        ->
        ((Float,String) -> (Float,String))

bind f' (gx,gs) = (fx, gs++fs)
   where (fx,fs) = f' gx
```

Podemos componerlas así:

```haskell
bind f' . g'
```

```haskell
bind :: (Float -> (Float,String)) -> ((Float,String) -> (Float,String)
bind f' (gx,gs) = (fx, gs++fs)
   where (fx,fs) = f' gx
```

Ejemplo:

```haskell
addDebug :: Float -> (Float, String)
addDebug x = (x+1, " (Add one) ")
mulDebug :: Float -> (Float, String)
mulDebug x = (x*2, " (Mul two) ")

composed :: Float -> (Float, String)
composed = bind addDebug . mulDebug

> composed 3
(7, " (Mul two) (Add one) ")
```

```haskell
g  :: Float -> Float
f' :: Float -> (Float, String)
```

Por último, queremos utilizar funciones no ”debuggeables”sin que muestren mensaje de error. Para eso definimos una función que transforme un resultado sin mensaje en uno con mensaje vacío:

```haskell
unit :: Float -> (Float, String)
unit x = (x,"")
```

De esta forma podemos encajar la función sin mensaje:

```haskell
bind f' . (unit . g)
```

Aplicación de todo:

```
1  addDebug :: Float -> (Float, String)
2  addDebug x = (x+1, " (Add one) ")
3  mulDebug :: Float -> (Float, String)
4  mulDebug x = (x*2, " (Mul two) ")
5
6  composed :: Float -> (Float, String)
7  composed = bind (bind addDebug . mulDebug) . (unit . (-1) )
8
9  > composed 3
10 (5, " (Mul two)   (Add one) ")
```

Las raíces cuadradas y cúbicas en espacio complejo pueden tener múltiples soluciones

```
1 sqrt',cbrt' :: Complex Float -> [Complex Float]
```

¿Cómo componemos funciones que devuelven múltiples valores?

```
1 sixrt' x = sqrt' (cbrt' x)
2 sixrt' = sqrt' . cbrt '
```

Error: no encajan los tipos. No se pueden componer.

```
sqrt',cbrt' :: Complex Float -> [Complex Float]
```

Nos definimos una nueva función para que encajen:

```
bind :: (Complex Float -> [Complex Float])
        ->
        ([Complex Float] -> [Complex Float])

bind f x = concat (map f x)
```

Podemos componerlas así:

```
sixrt' = bind sqrt' . cbrt'
```

```haskell
conjugate :: Complex Float -> Complex Float
sqrt' :: Complex Float -> [Complex Float]
```

Y otra función para transformar una funcion de un solo valor en otra con multiples resultados mediante composicion:

```haskell
unit :: Complex Float -> [Complex Float]
unit x = [x]
```

De esta forma podemos encajar la función de un solo parámetro en las de varios.

```haskell
bind sqrt' . (unit . conjugate)
```

```haskell
data Debuggable a = Writer (a, String)
data Multivariate a = Multi [a]
```

Ambas dos estructuras tienen un comportamiento común (mónada):

```haskell
bind :: (a -> monad a) -> monad a -> monad a
unit :: (a -> monad a)
```

Idea clave: El *estado* se convierte en un parámetro y un dato devuelto.

El concepto de **mónada** permite la inclusión de un tipo de dato que gestiona el *estado* del sistema. Las funciones que trabajan con mónadas pueden modificar dicho estado. Dicho estado es parámetro y dato devuelto en las funciones que interactúan con la mónada.

Una mónada se construye a partir de un tipo polimórfico que gestiona el *estado*.

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

    m >> k           =  m >>= \_ -> k
```

El tipo genérico *m* contiene la información de *estado*.

La función *do* se construye en base a la definición de una mónada:

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

Por eso funciona para entrada / salida.

```haskell
instance Monad (Debuggable a) where
  (>>=) :: Debuggable a -> (a -> Debuggable b) -> Debuggable b
  (>>=) (Writer (gx,gs)) f' = Writer (fx, gs++fs)
    where (fx,fs) = f' gx

  return  :: a -> Debuggable a
  return x = Writer (x,"")
```

```haskell
return 7 >>= (\x -> Writer (x+1,"inc."))
    >>= (\y -> Writer (2*y,"double."))
    >>= (\z -> Writer (z-1,"dec."))
```

o bien

```haskell
do
    let x = 7
    y <- Writer (x+1,"inc.")
    z <- Writer (2*y,"double.")
    Writer (z-1,"dec.")
```

da como resultado

```haskell
(15,"inc.double.dec.")
```

Entrada / salida

Entrada / salida

¿Cómo interactuamos con el usuario? (con los ficheros es de forma similar)
Haskell proporciona una serie de funciones que permiten escribir por la salida estándar (la terminal):

```
1 putChar  :: Char -> IO ()
2 putStr   :: String -> IO ()
3 putStrLn :: String -> IO ()  -- adds a newline
4 print    :: Show a => a -> IO ()
```

Ejemplo: imprime los primeros 20 enteros y sus cuadrados.

```
1 main = print ([(n, n^2) | n <- [0..19]])
```

También proporciona para leer por la entrada estándar:

```
1 getChar     :: IO Char
2 getLine     :: IO String
3 getContents :: IO String
4 interact    :: (String -> String) -> IO ()
5 readIO      :: Read a => String -> IO a
6 readLn      :: Read a => IO a
```

Ejemplo: elimina todos los caracteres que no son ASCII de la entrada estandar y los muestra por la salida estandar:

```
1 main = interact (filter isAscii)
```

Y se pueden secuenciar (una instrucción detrás de otra):

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

Ejemplo: el mismo que antes (pero entre ficheros)

```
readFile "input-file"                          >>= \ s ->
       writeFile "output-file" (filter isAscii s)  >>
       putStr "Filtering successful\n"
```

Para hacerlo más sencillo (legible) disponemos de la función *do*:

```
1  main = do
2          putStr "Input file: "
3          ifile <- getLine
4          putStr "Output file: "
5          ofile <- getLine
6          s <- readFile ifile
7          writeFile ofile (filter isAscii s)
8          putStr "Filtering successful\n"
```

La función *do* se construye en base a la definición de una mónada:

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

Por eso funciona para entrada / salida.

```haskell
main = do putStr "What is your name?"
          putStr "How old are you?"
          putStr "Nice day!"
```

es lo mismo que

```haskell
main = (putStr "What is your name?")
       >> ( (putStr "How old are you?")
            >> (putStr "Nice day!")
          )
```

Entrada / salida

```haskell
main = do a <- readLn
          print a
```

es lo mismo que

```haskell
main = readLn
       >>= (\a -> print a)
```

Ejemplos de mónadas

La mónada *Maybe* representa un valor que puede ser nulo.

```haskell
data Maybe a = Nothing | Just a

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

La mónada *State* representa un valor que puede almacenarse y recuperarse más adelante. Formalmente, representa un *estado* de una máquina de estados (como un lenguaje imperativo).

```
1 newtype State s a = State { runState :: (s -> (a,s)) }
2
3 instance Monad (State s) where
4     return a        = State $ \s -> (a,s)
5     (State x) >>= f = State $ \s -> let (v,s') =
6                                         x s in runState (f v) s'
```

Define una clase que permite almacenar y recuperar estados:

```haskell
class MonadState m s | m -> s where
    get :: m s
    put :: s -> m ()

instance MonadState (State s) s where
    get   = State $ \s -> (s,s)
    put s = State $ \_ -> ((),s)
```

Esto es especialmente útil para, por ejemplo, números pseudo-aleatorios.

- **[]**. Las listas también son una mónada para poder utilizarlas dentro de otras mónadas (en secuencias, entrada / salida).
- **Error** para gestión de errores.
- ...

Haskell Avanzado: Mónadas – I/O

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20