Haskell

June 30, 2015

1 Haskell

1.1 Introducción

- Diseñado por las universidades de Yale y Gasgow
- Su nombre es en homenaje a Haskell Curry
- Buscaba reunir las características de los lenguajes funcionales hasta el momento

Principalmente del lenguaje Miranda:

- Es un lenguaje orientado al uso comercial en vez de al uso académico (uno de los objetivos principales del diseño de Miranda)
- Lenguaje puramente funcional (se explica a continuación)
- Evaluación perezosa. Los argumentos se pasan a las funciones sin evaluar.

1.1.1 Lenguaje puramente funcional

Funcional: - Enfatiza la aplicación de funciones. - No maneja datos mutables o de estado.

Puro (funciones puras): - No hay efectos secundarios (para una misma entrada espero siempre el mismo resultado)

Más características:

- Tipado fuerte (datos de tipo concreto)
- Tipado estático (comprobación de tipos durante la compilación)
- Inferencia de tipos
- Muy alto nivel

1.2 Compiladores / intérpretes

Implementación utilizada: GHC (Glasgow Haskell Compiler)

- Escrito en Haskell, C y C++
- Disponible para varias plataformas (Windows, Mac OS X, y la mayoría de los sistemas UNIX) y mayoría de arquitecturas de procesador.
- Compila a código nativo
- Posee un intérprete (GHCi)
- Soporta concurrencia y paralelismo.
- Posee una gran cantidad de librerías, aunque algunas sólo funcionan bajo GHC.

Otras implementaciones:

- Hugs (intérprete)
- nhc98 (compilador)
- HBC (compilador)

2 Primeras funciones

2.1 Bien básico:

```
In [1]: doubleMe x = x + x

        doubleMe 4
```

```
8
```

```
In [2]: doubleUs x y = x*2 + y*2

        doubleUs 5 6
```

```
22
```

2.1.1 Sentencia if then else

```
In [3]: doubleSmallNumber x = if x > 100
                              then x
                              else x*2

        doubleSmallNumber 40
        doubleSmallNumber 200
```

```
80


200
```

2.2 Listas

```
In [ ]: numerosPrimosChicos = [1,2,3,5,7]
```

Concatenación

```
In [4]: numerosPrimosChicos = [1,2,3,5,7]
        numerosParesChicos = [2,4,6,8]

        numerosPrimosChicos ++ numerosParesChicos
```

```
[1,2,3,5,7,2,4,6,8]
```

Las cadenas de caracteres son listas de caracteres

```
In [5]: nombre = "Haskell "
        apellido = "Curry"

        nombre ++ apellido
```

```
"Haskell Curry"
```

```
In [6]: listaDeTuplas = [('b','a','z'),('x','f','d')]
```

2.2.1 Algunas funciones para listas

```
In [7]: length [5,4,3,2,1]
```

```
5
```

```
In [8]: reverse [5,4,3,2,1]
```

```
[1,2,3,4,5]
```

```
In [9]: null [1,2,3]
```

```
False
```

```
In [10]: take 4 [1,2,3,4,5,6,7]
```

```
[1,2,3,4]
```

```
In [11]: minimum [8,4,2,1,5,6]
```

```
1
```

```
In [12]: maximum [8,4,2,1,5,6]
```

```
8
```

```
In [13]: sum [5,2,1,6,3,2,5,7]
```

```
31
```

```
In [14]: product [5,2,1,6,3,2,5,7]
```

```
12600
```

3 Tipos

3.1 Tipado

Fuerte
  Estático

3.2 Tipos básicos

3.2.1 Enteros acotados (Int):

De -2147483648 a 2147483647 en máquinas de 32 bits

3.2.2 Enteros no acotados (Integer):

Menos eficientes que Int

3.2.3 Números reales de punto flotante de precisión simple (Float):

```
In [15]: circumference :: Float -> Float
         circumference r = 2 * pi * r
         circumference 4.0
```

```
25.132742
```

3.2.4 Números reales de punto flotante de precisión doble (Double):

```
In [16]: circumference :: Double -> Double
         circumference r = 2 * pi * r
         circumference 4.0
```

```
25.132741228718345
```

3.2.5 Booleanos (Bool):

True o False

3.2.6 Caracteres (Char):

```
In [17]: :t 'a'
```

```
'a' :: Char
```

3.3 Tipos en funciones

```
In [18]: sumarDos :: Int -> Int
         sumarDos a = 2 + a

         sumarDos 4
         :t sumarDos
```

```
6


sumarDos :: Int -> Int
```

```
In [19]: addThree :: Int -> Int -> Int -> Int
         addThree x y z = x + y + z

         addThree 1 2 3
         :t addThree
```

```
6


addThree :: Int -> Int -> Int -> Int
```

3.4 Variables de tipo

Una variable de tipo equivale a un tipo genérico.

Funciones con variables de tipo => Funciones polimórficas

Ejemplos:

```
In [20]: :t head
```

```
head :: forall a. [a] -> a
```

```
In [22]: fst (1, 2)
         :t fst
```

```
1
```

```
fst :: forall a b. (a, b) -> a
```

3.5 Clases de tipo

```
In [23]: 5 == 4

         (==) 'a' 'a'
```

```
False
```

```
True
```

¿Pueden tener cualquier tipo? No. Tienen que ser tipos comparables.

```
In [24]: :t (==)
```

```
(==) :: forall a. Eq a => a -> a -> Bool
```

(Eq a) es una restricción en la declaración de tipos.
Eq es una CLASE DE TIPO -> Funciona como interfaz. Nos dice que el tipo genérico "a" es comparable.

3.5.1 Otras clases de tipo

Ord -> Para tipos que poseen algún orden.

```
In [25]: :t (>)
```

```
(>) :: forall a. Ord a => a -> a -> Bool
```

Show -> Para tipos que pueden ser representados por cadenas

```
In [26]: show 3
         show False
         :t show
```

```
"3"
```

```
"False"
```

```
show :: forall a. Show a => a -> String
```

4 Sintaxis de funciones

4.1 Ajuste de patrones (Pattern Matching)

```
In [27]: sayMe :: (Integral a) => a -> String
         sayMe 1 = "Uno!"
         sayMe 2 = "Dos!"
         sayMe 3 = "Tres!"
         sayMe x = "No entre uno 1 y 3"

         sayMe 2
         sayMe 5
```

```
"Dos!"
```

```
"No entre uno 1 y 3"
```

¿Cuál es la ventaja? No necesito ir anidando if, then, else.

4.1.1 Más ejemplos

Un caso que falla

```
In [28]: charName :: Char -> String
         charName 'a' = "Alejandro"
         charName 'b' = "River"
         charName 'c' = "Carlos"

         charName 'a'
         charName 'b'
         charName 'c'
         charName 'z'
```

```
"Alejandro"
```

```
"River"
```

```
"Carlos"
```

```
<interactive>:(2,1)-(4,23): Non-exhaustive patterns in function charName
```

Utilizando tuplas

```
In [29]: addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
         addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

         addVectors (3,6) (2,4)
```

```
(5,10)
```

```
In [ ]: first :: (a, b, c) -> a
        first (x, _, _) = x

        second :: (a, b, c) -> b
        second (_, y, _) = y

        third :: (a, b, c) -> c
        third (_, _, z) = z
```

Con listas intencionales

```
In [30]: xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]

         [a+b | (a,b) <- xs]
```

```
[4,7,6,8,11,4]
```

Utilizando recursividad y el patrón x:xs (muy utilizado)

```
In [31]: length' :: (Num b) => [a] -> b
         length' [] = 0
         length' (_:xs) = 1 + length' xs

         length' [1,2,3,4,5]
```

```
5
```

Renombrando al patrón

```
In [32]: capital :: String -> String
         capital "" = "¡Una cadena vacía!"
         capital cadena@(x:_) = "La primera letra de " ++ cadena ++ " es " ++ [x]

         capital "Hola"
```

```
"La primera letra de Hola es H"
```

4.2 Guardas

```
In [35]: comparar :: (Ord a) => a -> a -> String
         comparar a b
             | a > b = "Primero mayor"
             | a < b = "Segundo mayor"
             | a == b = "Iguales"

         comparar 5 8
         comparar 10 8
         comparar 8 8
```

```
"Segundo mayor"


"Primero mayor"
```

```
"Iguales"
```

4.2.1 Where

Construcciones sintácticas que permiten ligar las variables al final de la función para que toda la función acceda

```
In [36]: iniciales :: String -> String -> String
         iniciales nombre apellido = n : [a]
             where (n:_) = nombre
                   (a:_) = apellido

         iniciales "Haskell" "Curry"
```

```
"HC"
```

4.2.2 Let

Expresiones que permiten ligar las variables en cualquier lugar. Siguen siendo locales.
Su forma es let {definición} in {expresión}

```
In [38]: cylinder :: (RealFloat a) => a -> a -> a
         cylinder r h =
             let areaLateral = 2 * pi * r * h
                 areaTapa = pi * r ^2
             in areaLateral + 2 * areaTapa
```

4.2.3 Case

Su forma es:

```
case expresion of patron -> resultado
                  patron -> resultado
                  patron -> resultado
```

```
In [39]: head' :: [a] -> a
         head' xs = case xs of [] -> error "¡head no funciona con listas vacías!"
                               (x:_) -> x

         head' [1, 2, 3]
```

```
1
```

4.3 Inferencia de tipos

Qué parte de Hindley-Milner no se entiende?
Todo!!!

```
In [40]: sumar x y = x + y
         :t sumar
```

```
sumar :: forall a. Num a => a -> a -> a
```

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad [\texttt{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'} \quad [\texttt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\ x\ .\ e : \tau \to \tau'} \quad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau} \quad [\texttt{Let}]$$

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \quad [\texttt{Inst}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall\ \alpha\ .\ \sigma} \quad [\texttt{Gen}]$$

Figure 1: caption

sumar x y = (+) x y
e0 = (+) x
e1 = y
e0 : $\tau \to \tau$'
e1 : τ'
e'0 = (+)
e'1 = x
e'0 : τ1 $\to$ t1'
e'1 : τ1
e0 : τ'1 = $\tau \to \tau$'
como (+) : (Num a) => a $\to$ a $\to$ a
τ1 $\to \tau \to \tau$' = a $\to$ a $\to$ a
τ1 $\sqsubseteq$ a
$\tau \sqsubseteq$ a
t' $\sqsubseteq$ a
Finalmente solo resta recomponer todo, usando que sumar = e'0 e'1 e1 El tipo de sumar es (Num a) => a $\to$ a $\to$ a.

4.4 Recursión

Definición de una función incluye a la propia función.
Necesita un caso base

```
In [42]: factorial 0 = 1
         factorial n = n * factorial (n - 1)

In [43]: factorial 10
```

```
3628800
```

Muchas de las funciones comunes se definen recursivamente

```
In [ ]: length [] = 0
        length (x : xs) = 1 + length xs

        length [1, 2, 3]
```

La recursividad suele estar abstraida

```
In [44]: factorial n = product [1 .. n]
```

4.5 Pereza

ya va ...
Nada se evalúa hasta que sea directamente necesario

```
In [45]: let (a, b) = (length [1 .. 5], reverse "hola mundo") in 1 + 1
         b
```

```
2
```

```
Not in scope: ‘b’
```

Evaluación parcial

```
In [46]: let (a, b) = (length [1 .. 5], reverse "hola mundo")
             (x:xs) = b in x
         xs
```

```
'o'
```

```
[(1,3),(4,3),(2,4),(5,3),(5,6),(3,1)]
```

La desventaja de la evaluación perezosa es que es dificil de predecir el uso de memoria

```
In [ ]: 3 + 2 :: Int
        5 :: Int
```

4.6 Garbage Collection

Limpieza de la memoria

- Haskell genera mucha basura debido a la inmutabilidad
- Cada paso recursivo genera nuevos datos
- Generación de datos jovenes y viejos
- Mientras más datos jovenes basura - mejor!

4.7 Funciones de orden superior

Una función de orden superior es aquella que puede tomar funciones como parámetro, o devolver una función como resultado, o ambas cosas.

4.7.1 Funciones como parámetros

```
In [48]: map' :: (a->b)->[a]->[b]
         map' _ [] = []
         map' f (x:xs) = (f x):map' f xs
```

```
In [49]: map' (*2) [1,2,3]
         map' (map' (2*)) [[1,2,3 ],[9, 0, 54]]
```

```
[2,4,6]
```

```
[[2,4,6],[18,0,108]]
```

```
In [ ]: fQuickSort :: (Ord b) => (a -> b) -> [a] -> [a]
        fQuickSort _ [] = []
        fQuickSort f (x:xs) =
          let
              menores = fQuickSort f [a | a <- xs, (f a) <= (f x)]
              mayores = fQuickSort f [a | a <- xs, (f a)  > (f x)]
          in menores ++ [x] ++ mayores

        fQuickSort length ["pipo","locura","purrete","amigo mio", "si"]
```

```
In [ ]: qsrt f (x:xs) = [a | a <- xs, (f a) <= (f x)] ++ x:[a | a <- xs, (f a)  > (f x)]
        qsrt _ _ = []
```

```
In [ ]: norma (x,y) = sqrt (x*x + y*y)

        fQuickSort norma [(1,2),(3,1),(5,0),(1,0),(-2,-1)]
        qsrt norma [(1,2),(3,1),(5,0),(1,0),(-2,-1)]
```

4.7.2 Funciones como resultado

```
In [50]: multiplicarPor :: (Num a) => a -> (a -> a)
         multiplicarPor x = (*) x
```

```
In [51]: let x10 = multiplicarPor 10
             x7  = multiplicarPor 7
         x10 5
         x7 5
```

```
50


35
```

4.7.3 Funciones currificadas

```
In [ ]: let max4 = max 4
        max4 7
        max4 1
```

```
In [52]: sumar3 x y z = x + y + z
         :t sumar3
         :t sumar3 1
         :t sumar3 1 2
```

```
sumar3 :: forall a. Num a => a -> a -> a -> a


sumar3 1 :: forall a. Num a => a -> a -> a


sumar3 1 2 :: forall a. Num a => a -> a
```

4.7.4 Pliegues (folds)

Son patrones existentes para implementar iteraciones.

```
In [ ]: foldl' :: (a->b->a)-> a->[b]->a
        foldl' _ acc [] = acc
        foldl' f acc (x:xs) = foldl' f (f acc x) xs
```

```
In [ ]: elem' :: (Eq a)=> a -> [a] -> Bool
        elem' y ys = foldl' f False ys
          where f acc x = if x==y then True else acc
```

4.7.5 Funciones anónimas (lambdas)

Las creamos para pasarlas como parametros a funciones de orden superior (generalmente)

```
In [53]: filter (\(x,y)-> x >= 5) [(5,6),(5,4),(4,4),(8,1),(9,0),(9,2)]
```

```
[(5,6),(5,4),(8,1),(9,0),(9,2)]
```

```
In [54]: elem' :: (Eq a)=> a -> [a] -> Bool
         elem' y ys = foldl' (\acc x-> if x==y then True else acc) False ys
```

```
Line 2: Redundant if
Found:
if x == y then True else acc
Why not:
((x == y) || acc)


Not in scope: foldl'
Perhaps you meant one of these: ‘foldl1’ (imported from Prelude), ‘foldl’ (imported from Prelude), ‘fol
```

```
In [ ]: elem' 1 [1, 2, 3 ,4 ,5, 6]
        elem' 9 [1 ,2 ,3, 4 ,5 ,6]
        elem' 1 []
```

4.8 El operador “$”

Aplicación de función

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

```
In [55]: sum (filter (> 10) (map (*2) [2..10]))
```

```
80
```

```
In [56]: sum $ filter (> 10) $ map (*2) [2..10]
```

```
80
```

4.9 Composicion de funciones

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

```
In [57]: map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
```

```
Line 1: Avoid lambda
Found:
\ x -> negate (abs x)
Why not:
negate . abs


[-5,-3,-6,-7,-3,-2,-19,-24]
```

```
In [58]: map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
```

```
[-5,-3,-6,-7,-3,-2,-19,-24]
```

5 Input/Output

5.1 Mostrando por pantalla

```
In [59]: main = putStrLn "Hola mundo"

         main
```

```
Hola mundo
```

```
In [60]: :t putStrLn
         :t putStrLn "Hola mundo"
```

```
putStrLn :: String -> IO ()
```

```
putStrLn "Hola mundo" :: IO ()
```

Vemos que putStrLn "Hola mundo" devuelve una acción IO, la cual no devuelve valores -> ().
Si queremos tener más de una acción IO, las ejecutamos en el bloque "do".

```
In [62]: main = do
             putStrLn "Ingrese su nombre"
             nombre <- getLine
             putStrLn ("Hola " ++ nombre)
         main
```

```
Ingrese su nombre
Hola Esteban
```

```
In [63]: :t getLine
```

```
getLine :: IO String
```

getLine también es una acción IO, pero en este caso sí devuelve un valor, de tipo String.

5.1.1 Y esto de los tipos, ¿De qué nos sirve?

```
In [64]: main = do
             putStrLn "Ingrese su nombre"
             nombre <- getLine
             putStrLn ("Hola " ++ nombre)

         main
```

```
Ingrese su nombre
Hola Pablo
```

Sirve para separar código impuro de código puro.
Vemos si la siguiente acción es válida:

```
In [65]: main = "Hola, mi nombre es" ++ getLine
```

```
Couldn't match expected type 'String' with actual type 'IO String'
In the second argument of '(++)', namely 'getLine'
In the expression: "Hola, mi nombre es" ++ getLine
In an equation for 'main:main': main:main = "Hola, mi nombre es" ++ getLine
```

5.2 Otras acciones IO:

5.2.1 putStr

```
In [66]: do
             putStr "Hola, "
             putStr "programo "
             putStr "en "
             putStrLn "Haskell!"
```

```
Hola, programo en Haskell!
```

5.2.2 putChar

```
In [67]: do
             putChar 'F'
             putChar 'I'
             putChar 'U'
             putChar 'B'
             putChar 'A'
```

```
FIUBA
```

5.2.3 print

Muestra miembros de la clase show

```
In [68]: do
             print True
             print 2
             print "Hola"
             print 3.2
             print [3,4,3]
```

```
True
2
"Hola"
3.2
[3,4,3]
```

5.2.4 when

```
In [69]: import Control.Monad

         main = do
             c <- getChar
             when (c /= ' ') $ do
```

```
                putChar c
                main
        main
```

```
asdasdd
```

5.2.5 sequence

```
In [70]: do
             rs <- sequence [getLine, getLine, getLine]
             print rs
```

```
["","asd","asd"]
```

5.3 Aleatoriedad

5.3.1 Clases de tipo utilizadas

Están en el módulo System.Random

```
Random -> Las f pueden tener datos aleatorios
RandomGen -> Las f pueden generar datos aleatorios
```

```
In [71]: import System.Random
         :t random
```

```
random :: forall a g. (RandomGen g, Random a) => g -> (a, g)
```

```
In [73]: import System.Random
         :t random
```

```
random :: forall a g. (RandomGen g, Random a) => g -> (a, g)
```

```
In [72]: random (mkStdGen 100)
```

```
(-3650871090684229393,693699796 2103410263)
```

El primer valor es el valor que queremos. El segundo valor es un nuevo generador aleatorio. Otra vez:

```
In [74]: random (mkStdGen 100)
```

```
(-3650871090684229393,693699796 2103410263)
```

Dio el mismo valor. Cambiamos por otro generador:

```
In [75]: random (mkStdGen 2432434)
```

```
(708227736329069275,242756692 2103410263)
```

Cambiando el tipo:

```
In [76]: random (mkStdGen 949488) :: (Float, StdGen)

(0.8241101,1597344447 1655838864)


In [77]: random (mkStdGen 949488) :: (Bool, StdGen)

(False,1485632275 40692)
```

6 Módulos

- Son colecciones de funciones, tipos, y clases de tipos.
- No suelen depender de otros módulos.
- Se utilizan para dividir justamente el código (encapsulamiento), simplificando la programación.

6.1 Importando módulos

Se importan todas las funciones del módulo y pasan a estar en el espacio global.

Todo el módulo:

```
import Data.List
```

Una o más funciones del módulo:

```
import Data.List (intercalate, sort)
```

Todo el módulo excepto una o más funciones:

```
import Data.List hiding (intercalate, sort)
```

Un módulo que tiene funciones con el mismo nombre que otras de otros módulos importados:

```
import qualified Data.Map
```

Abreviando el módulo

```
import Data.List as M
```

6.2 Algunos módulos

Para listas:

```
Data.List
```

Para caracteres:

```
Data.Char
```

Para listas de asociación: listas de duplas (“clave”,“valor”)

```
Data.Map
```

Para conjuntos:

```
Data.Set
```

6.3 Creando módulos

Se define el módulo y dentro sus funciones (en este caso con la construcción where)

```
In [78]: module Areas
         ( areaCirculo
         , areaCuadrado
         , areaRectangulo
         ) where

         areaCirculo :: Float -> Float
         areaCirculo radio = pi * (radio ^ 2)

         areaCuadrado :: Float -> Float
         areaCuadrado lado = lado ^ 2

         areaRectangulo :: Float -> Float -> Float
         areaRectangulo a b = a * b
```

Para impotarlo en otro lado, se hace como ya habíamos visto:

```
In [79]: import Areas

         areaCirculo 1.0
         areaCuadrado 2.0
         areaRectangulo 3.0 1.0
```

```
3.1415927


4.0


3.0
```

- Areas está en el mismo directorio que donde estamos trabajando.

6.4 Functores

Contexto computacional

Contexto: computación podría tener un valor, o podría fallar

fmap

- Listas []
- Maybe
- Either a
- Tree (definido por usuario)
- IO

```
In [ ]: instance Functor IO where
            fmap f action = do
                result <- action
                return (f result)
```

- (->) r

```
In [ ]: instance Functor ((->) r) where
            fmap = (.)
```

6.4.1 Leyes de Functores

- fmap id = id
- (f . g) = fmap f

6.5 Functores Aplicativos

Aplicacion de una funcion encerrada en un functor a otro functor

```
In [ ]: class (Functor f) => Applicative f where
            pure :: a -> f a
            (<*>) :: f (a -> b) -> f a -> f b
```

6.6 Mónadas

Descripción componibles de computaciones

Permiten suplementar las funcionalidades puras con I/O, estado, indeterminismo, etc.
En terminos del lenguaje una mónada es un tipo instancia de la clase *Monad*.

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

Mónada se puede ver como un contenedor: *return* pone el valor **a** adentro de ella.
Operador *bind* aplica una función al contenido de la mónada devolviendo otra mónada

```
In [80]: import Control.Applicative

         putStrLn "Como te llamas?"
             >>= (\_ -> getLine)
             >>= (\name -> putStrLn ("Hola, " ++ name ++ "!"))
```

```
Line 2: Use const
Found:
\ _ -> getLine
Why not:
const getLine


Como te llamas?
Hola, Seba!
```

El operador (>>=) se ocupa de tomar el valor del lado izquierdo y combinarlo con la función del lado derecho para producir un valor nuevo

```
In [81]: do
             putStrLn "Como te llamas?"
             name <- getLine
             putStrLn ( "Hola " ++ name ++ "!" )
```

```
Como te llamas?
Hola Seba!
```

- Código imperativo

6.6.1 Las Leyes de las Mónadas

```
-- Identidad por la izquierda
return x >>= f = f x

-- Identidad por la derecha
m >>= return = m

-- Asociatividad
(m >>= f) >>= g = m >>= (x -> f x >>= g)

-- Operador Composición
(>=>) :: Monad m => ( a -> m b ) -> ( b -> m c ) -> a -> m c
(m >=> n ) x = do
    y <- m x
    n y

-- Identidad por la izquierda
return >=> f = f
-- Identidad por la derecha
f >=> return = f
-- Asociatividad
(f >=> g) >=> h = f >=> (g >=> h)
```

Mónada	Semántica imperativa
Maybe	Excepción anónima
Error	Excepción con descripción
State	Estado global
IO	Entrada y Salida
[] (lista)	Indeterminismo
Reader	Entorno
Writer	Logger

Maybe

```
data Maybe a = Nothing | Just a

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

Útil para encadenar computaciones que pueden devolver Nothing como resultado
Either

```
In [ ]: data Either a b = Left a | Right b
```

Lista

```
instance Monad [] where
    m >>= f = concat (map f m)
    return x = [x]
```

```
In [82]: do
             x <- [1 .. 10]
             y <- [2, 3, 5, 7]
             return (x * y)
```

```
[2,3,5,7,4,6,10,14,6,9,15,21,8,12,20,28,10,15,25,35,12,18,30,42,14,21,35,49,16,24,40,56,18,27,45,63,20,3
```

```
In [83]: [ x * y | x <- [1 .. 10], y <- [2, 3, 5, 7] ]
```

```
[2,3,5,7,4,6,10,14,6,9,15,21,8,12,20,28,10,15,25,35,12,18,30,42,14,21,35,49,16,24,40,56,18,27,45,63,20,3
```

State

```
In [85]: newtype State s a = State {runState :: (s -> (a, s))}

         instance Monad ( State s ) where
             return x = State $ \s -> ( x, s )
             ( State h ) >>= f = State $ \s -> let ( a, newState ) = h s
                                                   ( State g ) = f a
                                               in g newState
```

```
In [86]: import Control.Monad

         type Stack = [Int]

         pop :: State Stack Int
         pop = State $ \(x:xs) -> (x, xs)

         push :: Int -> State Stack ()
         push a = State $ \xs -> ((), a : xs)

         do
             push 3
             pop
```

```
No instance for (Show (State Stack Int)) arising from a use of ‘print’
Possible fix: add an instance declaration for (Show (State Stack Int))
In a stmt of an interactive GHCi command: print it
```

6.7 Zippers

Movimiento eficiente en estructuras

```
In [88]: data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
         data Crumb a = LeftCrumb a (Tree a) | RightCrumb a (Tree a) deriving (Show)
         type Breadcrumbs a = [Crumb a]

         type Zipper a = (Tree a, Breadcrumbs a)
```

```
In [89]: goUp :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
         goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
         goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

```
In [91]: goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
         goRight (Node x l r, bs) = (r, RightCrumb x l:bs)

         goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a  )
         goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)

In [92]: let tree = Node 1 (Node 2 Empty (Node 0 Empty Empty)) Empty

In [93]: goLeft (tree, [])
         goRight $ goLeft (tree, [])
         goUp $ goRight $ goLeft (tree, [])
```

```
(Node 2 Empty (Node 0 Empty Empty),[LeftCrumb 1 Empty])


(Node 0 Empty Empty,[RightCrumb 2 Empty,LeftCrumb 1 Empty])


(Node 2 Empty (Node 0 Empty Empty),[LeftCrumb 1 Empty])
```

```
In [94]: type ListZipper a = ([a],[a])

         goForward :: ListZipper a -> ListZipper a
         goForward (x:xs, bs) = (xs, x:bs)

         goBack :: ListZipper a -> ListZipper a
         goBack (xs, b:bs) = (b:xs, bs)

In [95]: goBack $ goForward $ goForward ([3,5..15],[])
```

```
([5,7,9,11,13,15],[3])
```

6.8 Concurrencia

6.8.1 Concurrencia vs paralelismo

- Varias tareas vs una sola
- Indeterminismo

```
In [96]: import Control.Concurrent (forkIO)
         import qualified Data.ByteString.Lazy as L
         import Codec.Compression.GZip (compress)

         compressFile = L.writeFile "files/foo.gz" . compress

         do
             content <- L.readFile "files/foo.txt"
             forkIO (compressFile content)
             putStrLn "Gracias por comprimir!"
             return ()
```

```
Gracias por comprimir!
```

6.8.2 Comunicación entre threads

Comunicación simple
Se utiliza Mvar - una variable sincronizable
Se puede ver como una caja con un valor

```
In [97]: import Control.Concurrent

         do
           m <- newEmptyMVar
           forkIO $ do
             v <- takeMVar m
             putStrLn ("Recibido " ++ show v)
           putStrLn "Enviando"
           putMVar m "Hola!"
```

```
Enviando
Recibido "Hola!"
```

Usar putMVar y takeMVar resulta en problemas comunes en concurrencia: deadlocks
modifyMVar es una combinación de los dos **segura**
Comunicación por canales

```
In [98]: import Control.Concurrent
         import Control.Concurrent.Chan

         do
           ch <- newChan
           forkIO $ writeChan ch "Hola!"
           forkIO $ writeChan ch "Hola desde otro hilo!"
           readChan ch >>= print
           readChan ch >>= print
```

```
"Hola!"
"Hola desde otro hilo!"
```

6.8.3 Paralelismo

Por defecto ghc usa un solo núcleo
Tenemos que pasar el parametro *-threaded* a la hora de linkear (creación del ejecutable)
Desventaja: mayor costo de creación de threads y manejo de MVars

```
In [99]: import Control.Parallel (par)

         parallelMap :: (a -> b) -> [a] -> [b]
         parallelMap f (x:xs) = let r = f x
                                in r `par` r : parallelMap f xs
         parallelMap _ _      = []

         parallelMap (*2) [1,2,4,3,5,8,190]
```

```
[2,4,8,6,10,16,380]
```

par Evalua el valor del lado izquierdo y devuelve el del lado derecho. Hace la evaluación en paralelo.

6.9 Software transactional memory

Mecanismo que trata de solucionar los problemas del modelo concurrente común.

Un bloque de acciones se ejecuta como una transacción. - Si otros threads no modifican la misma data que ese bloque -> exito - Sino el bloque se descarta y se vuelve a ejecutar

atomicaly es el bloque. *STM* es una mónada parecida a IO (tiene efectos secundarios).

```
In [100]: import Control.Concurrent.STM

          :t atomically

atomically :: forall a. STM a -> IO a
```

TVar son contenedores mutables parecidos a MVar, pero solo se pueden usar en una acción STM.

```
In [101]: type Account = TVar Int

          withdraw :: Account -> Int -> STM ()
          withdraw acc amount = do
              bal <- readTVar acc
              writeTVar acc (bal - amount)

          deposit :: Account -> Int -> STM ()
          deposit acc amount = withdraw acc (- amount)

          atomically $ do
              acc <- newTVar (20 :: Int)
              deposit acc 20
              withdraw acc 10
              readTVar acc

30
```

Retry permite reiniciar una transacción desde 0 (sin realizar todas las modificaciones)

```
In [103]: limitedWithdraw :: Account -> Int -> STM ()
          limitedWithdraw acc amount = do
              bal <- readTVar acc
              if amount > 0 && amount > bal
              then retry
              else writeTVar acc (bal - amount)
```

orElse permite hacer una acción en el caso de que otra falle (sea reiniciada)

```
In [104]: limitedWithdraw2 :: Account -> Account -> Int -> STM ()
          limitedWithdraw2 acc1 acc2 amt
            = orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

6.10 Conclusiones

6.10.1 ¿Preguntas?

6.11 Gracias!