Haskell

May 22, 2015

1 Recursión

La recursión o recursividad es la forma de especificar una función basandose en su propia definición. Es una parte muy importante de Haskell. Una función es recursiva cuando una parte de su definición incluye a la propia función. Necesita por lo menos un *caso base* que no hace llamado recursivo para que exista una condición límite.

Este ejemplo muestra una función de factorial recursiva, separando claramente el caso base.

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Muchas de las funciones comunes en Haskell se pueden definir de forma recursiva, por ejemplo el *length*, la función que devuelve el número de elementos de una lista:

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

Recursión es usada para definir casi todas las funciones de manejo de números y listas, sin embargo en la práctica no es de usarse tan seguido: la recursividad está abstraida por las funciones de las librerias de Haskell, permitiendo al programador razonar sus problemas en más alto nivel. Por ejemplo, la función de factorial de ejemplo escrita anteriormente se puede definir de la siguiente manera:

```
factorial n = product [1..n]
```

2 Pereza

Una característica destacable de Haskell es que es *perezoso* (o *no estricto*). Esto significa que nada se va a evaluar hasta que sea directamente necesario - la evaluación queda diferida hasta que el resultado es requerido por otra computación.

El siguiente es un ejemplo de la pereza de Haskell:

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

Como la expresión después del *in* no utiliza los valores de a y b, estos quedan sin evaluar, ya que no son necesarios. Tambien pueden no ser necesarios por completo, o ser *parcialmente* requeridos:

```
let (a, b) = (length [1..5], reverse "hola mundo")
    'o':ss = b
```

Como solo es necesaria la primera letra para concluir el matcheo de patrones, la evaluación es parcial.

Las funciones en Haskell pueden ser perezosas o estrictas. Podemos analizar si una función es perezosa o estricta pasandole **undefined** y viendo si su ejecución falla. Esto se debe a que en Haskell la evaluación forzada del **undefined** siempre termina en un error.

```
let (x, y) = undefined in x  -- Error!
length [undefined, undefined, undefined]
-- No hay error, length es perezoso
```

La evaluación perezosa tiene muchas ventajas, como la reutilización de código, posibilidad de generar estructuras de datos infinitas y definiciones circulares, pero su mayor inconveniente es que el uso de memoria se vuelve muy dificil de predecir, por ejemplo las expresiones 3+2 :: **Int** y 5 :: **Int** denotan el mismo valor pero pueden tener diferentes tamaños en memoria.

3 Mónadas

Las **mónadas** en Haskell se pueden pensar como descripciones *componibles* de computaciones. Presentan la posibilidad de separar la combinación de computaciones de su ejecución y permiten acarrear datos extra implícitamente en adición al resultado de la computación, que *se producirá* cuando la mónada sea corrida. De esta manera permiten suplementar las funcionalidades *puras* con I/O, estado, indeterminismo, etc.

En terminos del lenguaje una mónada es un tipo parametrizado que es instancia de la clase *Monad*. Su definición es la siguiente:

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

Podemos ver la mónada como un contenedor para un valor **a**. La función **return** se ocupa de poner ese valor adentro de la mónada. Entonces la función (>>=), tambien conocida como *bind*, aplica la función que se le pasa por parámetro al contenido de la mónada obteniendo como resultado otra mónada (obviamente la función pasada tiene que tener el tipo adecuado). Se puede ver como funciona en el siguiente ejemplo:

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

El operador (>>=) se ocupa de tomar el valor del lado izquierdo y combinarlo con la función del lado derecho para producir un valor nuevo. El ejemplo de arriba se puede reescribir con la notación **do**, que es un azucar sintáctico alrededor del operador *bind*:

```
do
    putStrLn "Como te llamas?"
    name <- getLine
    putStrLn ("Hola, " ++ name ++ "!")
```

Ese código puede parecer de un lenguaje imperativo, y de hecho lo es: otra forma de ver las mónadas es pensar que son la abstracción necesaria para suplementar las funcionalidades que no cuadran adentro del paradigma funcional.

La implementación mas sencilla del (>>=). toma el valor del lado izquierdo, le aplica la función y devuelve el resultado, sin embargo se vuelve realmente útil cuando esa implementación hace algo extra.

3.1 Las Leyes de las Mónadas

Las mónadas por convención deben cumplir las siguientes leyes:

```
-- 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)
```

Para poder entenderlo mejor y usarlo de una forma más sencilla, el módulo Control.Monad define el operador de composición de mónadas:

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

Con el uso de ese operador las tres reglas pueden escribirse de la siguiente forma:

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

```
-- Identidad por la derecha
f >=> return = f

-- Asociatividad
(f >=> g) >=> h = f >=> (g >=> h)
```

3.2 Mónadas comunes

La siguiente tabla lista las mónadas más comunes usadas en Haskell, denotando el problema que tratan de solucionar en términos imperativos.

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

En las siguientes secciones se explicarán algunas de las mónadas que aparecen en esta tabla.

3.3 Mónada Maybe

Es una de las mónadas más utilizadas y es muy sencilla a la vez. La definición de esta Mónada es la siguiente:

```
data Maybe a = Nothing | Just a

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

La mónada Maybe incorpora la posibilidad de encadenar computaciones que pueden devolver Nothing como resultado, en cuyo caso la cadena terminaría antes.

3.3.1 Either

Otra mónada muy usada es *Either*. Su tipo está definido de la siguiente manera:

```
data Either a b = Left a | Right b
```

Su propósito es muy parecido al de Myabe, Left es considerado un error mientras que Right - un valor normal. La diferencia es que Left permite guardar un valor (a diferencia del Nothing en el caso de Maybe).

3.4 Mónada Error

La mónada Error es la forma que tiene Haskell de simular las *excepciones*. Se logra de la siguiente forma: se encadenan computaciones que pueden lanzar excepción derivando la ejecución a la instancia que puede manejarla.

3.5 Mónada List

Resulta que la lista es una mónada tambien! Las listas se utilizan para modelar las computaciones no determinísticas que pueden devolver un número de resultados arbitrario. Se define de la siguiente manera:

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

El return es fácil de entender: simplemente devuelve una lista de un elemento. El operador *bind* tambien se entiende si consideramos que tiene que cumplir con el tipo [a] -> (a -> [b]) -> [b] (por su defenición más general). La función **f** aplica a cada elemento y, dado su tipo, retorna una lista - por eso es necesario el **concat** al final, para a la salida obtener una lista.

¿Por qué las listas son mónadas? Se explica con el siguiente ejemplo de notación monádica de una lista:

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

foo es un múltiplo de x e y, con x siendo un número no determinístico entre 1 y 10, e y siendo 2, 3, 5 o 7. Este ejemplo es la notación larga y monádica de una comprensión de lista, podría ser escrita así:

```
  foo = [x * y | x <- [1 .. 10], y <- [2, 3, 5, 7]]
```

3.6 Mónada IO

Dado que Haskell es un lenguaje funcional y perezoso, no podemos expresar las efectos reales de las operaciones de entrada y salida con funciones puras. De hecho, estas operaciones no se pueden ejecutar perezosamente, ya que esto haria que los efectos reales sean impredecibles. La mónada **IO** es el medio para representar las acciones de entrada/salida como valores de Haskell, para que el programador pueda manipularlos con funciones puras.

3.7 Mónada State

La mónada *State* permite acarrear estado a lo largo de una ejecución. Con ella se puede hacer lo siguiente: dado un valor de estado, se produce un resultado y un nuevo valor de estado. Esta es su definición:

```haskell
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
```

Como se ve en la declaración de tipo, State es solo una abstracción de una función que toma un estado, devuelve un valor intermedio y un nuevo estado.

Un ejemplo de uso de State es una pila. La función **push** agrega un elemento al tope de la pila y **pop** saca uno. Sin State tendriamos que arrastrar la pila como argumento a estas funciones, lo cual no es lo más cómodo de usar. Con State la podemos definir de la siguiente forma:

```haskell
type Stack = [Int]

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

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

Un ejemplo de uso de la pila sería:

```haskell
stackManip = do
    push 3
    a <- pop
    pop
```

La implementación de State en Control.Monad también es instancia de la clase MonadState que define 2 funciones muy útiles:

```haskell
put newState = State $ \_ -> ((), newState)
get = State $ \st -> (st, st)
```

Las cuales nos permiten manejar el estado de una manera más sencilla.

3.8 Concurrencia

Un programa concurrente necesita realizar varias tareas al mismo tiempo. Estas tareas no necesariamente tienen que estar relacionadas entre si. El correcto funcionamiento de un programa concurrente no necesita varios núcleos.

En contraste, un programa paralelo soluciona un solo problema con el mejor rendimiento posible, empleando para eso más de un núcleo.

3.8.1 Threads

Un hilo es una acción *IO* que se ejecuta independientemente de los otros hilos. Los hilos en Haskell no son determinísticos. Para crear un thread, usamos la función *forkIO* del módulo *Control.Concurrent*.

Un ejemplo de uso podría ser la compresión de un archivo

3.9 Paralelismo

References

[1] ¡Aprende Haskell por el bien de todos!, http://aprendehaskell.es

[2] Real World Haskell, http://book.realworldhaskell.org

[3] Recursion, http://en.wikibooks.org/wiki/Haskell/Recursion

[4] Recursion Patterns, https://www.fpcomplete.com/school/starting-with-haskell/introduction-to-haskell/3-recursion-patterns-polymorphism-and-the-prelude

[5] Laziness, http://en.wikibooks.org/wiki/Haskell/Laziness

[6] Understanding Monads, http://en.wikibooks.org/wiki/Haskell/Understanding_monads

[7] Monad, https://wiki.haskell.org/Monad

[8] What is a monad?, http://stackoverflow.com/questions/44965/what-is-a-monad

[9] Monad laws, https://wiki.haskell.org/Monad_laws

[10] All about monads, https://wiki.haskell.org/All_About_Monads

[11] The State Monad: A Tutorial for the Confused?, http://brandon.si/code/the-state-monad-a-tutorial-for-the-confused/