Mónadas

Mario Román

<2016-12-24 Sat 15:09>

> A monad is just a monoid in the category of endofunctors, what's the problem?
>
> – **Philip Walder** (apócrifa) en *A Brief, Incomplete, and Mostly Wrong History of programming languages*

Prerrequisitos

Este artículo requiere un conocimiento previo de Haskell, o al menos, de otro lenguaje de programación funcional. Puedes consultar nuestra introducción a Haskell con recursos para iniciarte en el lenguaje.

Por otro lado, para la segunda parte del artículo es recomendable conocimiento previo sobre teoría de categorías. Aun así, no es necesaria para leer la primera parte del artículo, donde hablamos de mónadas sin hacer ninguna referencia explícita a la teoría de categorías. Si quieres leer sobre ese tema, puedes consultar nuestros apuntes de introducción a teoría de categorías.

Motivación para las mónadas

Mónadas en Haskell

Imaginemos que necesitamos controlar cuando una función interna devuelve un error, o cuando usa un estado que debe ser pasado al resto de funciones. Cuando trabajamos con programación funcional pura, debemos devolver explícitamente el error (señalar cómo va a tratarlo cada función) o pasar el estado como argumento a cada una de las funciones; así que una solución sería modificar cada una de las funciones que usamos para que tenga en cuenta ese estado o ese caso de error, pero esto añadiría mucha complejidad innecesaria a nuestro código. La estructura de mónada simplifica esta escritura.

El siguiente ejemplo, en el que tratamos el manejo de errores encapsulado en una mónada, está inspirado en los ejemplos de:

- Monads for functional programming - *Philip Wadler*

Que es una muy buena introducción al uso de las mónadas en programación funcional.

Calculando raíces cuadradas

Por ejemplo, supongamos que intentamos sacar raíces cuadradas en los reales usando el método de Newton. Si intentamos calcular $\sqrt{n}$, podemos tomar a cada paso la aproximación:

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{n}{x_k}\right)$$

Y parar cuando estemos suficientemente cerca (más cerca que un dado):

$$|x_{k+1} - x_k| < \varepsilon$$

Escribimos una implementación de ese concepto de convergencia para listas infinitas en Haskell y del método de Newton, basado en la función `iterate`:

```
limit :: Float -> [Float] -> Float
limit epsilon (x:y:xs)
  | abs (x-y) < epsilon = y
  | otherwise           = limit epsilon (y:xs)

newtonaprox :: Float -> Float -> Float
newtonaprox n x = (x + n/x)/2

sqroot :: Float -> Float
sqroot 0 = 0
sqroot x = limit 0.03 (iterate (newtonaprox x) x)
```

Ahora imaginemos que usamos esta raíz cuadrada recién definida para solucionar una ecuación de segundo grado $x^2 + bx + c = 0$:

$$x = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$$

Podemos definir una estructura de datos `QPol` para el polinomio y una función que lo resuelva obteniendo sus dos raíces:

```
data QPol = QPol Float Float Float

instance Show QPol where
    show (Qpol a b c) = show a ++ "x + " ++ show b ++ "x + " show c

solve :: QPol -> (Float,Float)
solve (QPol a b c) = (sol1 sol2)
    where sol1 = ((-b) + sqroot(b*b-4*c*a))/(2*a)
          sol2 = ((-b) - sqroot(b*b-4*c*a))/(2*a)
```

Y podemos comprobar que funciona:

```
> pol = QPol 1 (-5) 6
> putStrLn $ "Las soluciones de " ++ show pol ++ " son " ++ solve pol
Las soluciones de 1.0x + -5.0x + 6.0 son (3.0,2.0)
```

Controlando los errores

Pero ¿qué ocurre cuando intentamos calcular la raíz cuadrada de un número no positivo? Este método no la encuentra, por lo que debería devolver un error antes de intentar empezar a calcularla. La solución obvia es reflejar este error con un `Maybe` en el cálculo de la raíz cuadrada.

```
sqroot' :: Float -> Maybe Float
sqroot' x
  | x < 0     = Nothing
  | x == 0    = Just 0.0
  | otherwise = Just ( limit 0.03 (iterate (newtonsqrt x) x) )
```

Esto lo soluciona, pero nos crea un problema mayor. La función `solve` está usando la raíz cuadrada y se espera de ella que devuelva un número, no un posible error. Si queremos conseguir que funcione con la nueva `sqroot'`, necesitaríamos implementar todas sus componentes internas teniendo en cuenta ese error. Por ejemplo, deberíamos reescribir el `(+)`, para tener en cuenta errores y propagarlos por todos los cálculos involucrando a `sqroot'`:

```haskell
(+.) :: Maybe Float -> Maybe Float -> Maybe Float
(+.) Nothing _ = Nothing
(+.) _ Nothing = Nothing
(+.) (Just a) (Just b) = Just (a + b)
```

Pero esto es muy pesado de implementar; deberíamos implementarlo ¡para cada una de las operaciones que usen la raíz cuadrada en algún punto! Esto obliga a cada una de nuestras operaciones intermedias a ser conscientes de la posibilidad de error, dándonos código mucho menos modular y reusable.

Una solución ligeramente mejor es la de abstraer este proceso de hacer a una función consciente de la posibilidad de error en una función aparte y definir las demás en función suya:

```haskell
errorAware :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
errorAware op Nothing _ = Nothing
errorAware op _ Nothing = Nothing
errorAware op (Just a) (Just b) = Just (op a b)

(+.), (*.) :: Maybe Float -> Maybe Float -> Maybe Float
(+.) = errorAware (+)
(*.) = errorAware (*)
```

Esto nos permite hacer cálculos con ellas:

```
> sqroot' (-3) +. Just 4
Nothing
> sqroot' 3 +. Just 4
Just 5.732143
```

La mónada Maybe

Esta idea para simplificar el tratamiento de errores, realizada correctamente, es lo que nos va a proporcionar la estructura de mónada. En Haskell, podemos definir una mónada como:

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

La idea intuitiva es que `(>>=)` nos permite tomar una función que puede devolver errores pero que no comprueba a la entrada si ha recibido un error, es decir, de tipo `(a -> Maybe b)` ; y aplicarla sobre una función que puede contener un error. La podríamos usar por ejemplo para componer varias `sqroot'`, que era algo que hasta ahora no podíamos hacer sin tratar cada posible caso de error. Y la función `return`, que en este caso es simplemente `Just`, nos permite considerar una constante como un posible error. Podemos calcular fácilmente así

$$\sqrt{\sqrt{3}}$$

teniendo en cuenta los casos de error:

```
sqroot' (sqroot' 3)                -- ¡Error de tipos!
sqroot' 3 >>= sqroot'              -- Usando mónadas
Just   3 >>= sqroot' >>= sqroot'   -- Usando Just
return 3 >>= sqroot' >>= sqroot'   -- Equivalente a lo anterior
```

Notación do

Las mónadas definen las funciones anteriores y muchas más que no vamos a tratar ahora mismo, pero como resultado, nos acaban ofreciendo la **notación do**, que es la que podemos usar para acabar escribiendo nuestra función `solve` como:

```
solve :: QPol -> Maybe (Float,Float)
solve (QPol a b c) = do
  discriminant <- sqroot' (b*b - 4*c*a)
  return (((-b) + discriminant)/(2*a), ((-b) - discriminant)/(2*a))
```

En la primera línea tenemos en cuenta que la función `sqroot` puede producir error, y en la segunda simplemente usamos el *posible* resultado de ella sin tener que preocuparnos por el resto de funciones.

Nótese que la notación **do** es sólo una notación diseñada para aliviar la escritura de operaciones con mónadas en algunos casos particulares; es sólo *azúcar sintáctico* para operaciones que no dejan de ser puramente funcionales. Existen críticas al uso de esta notación. [1]

Mónadas en programación funcional

Mónadas como clase de tipos

Las mónadas en Haskell están definidas como una clase de tipos teniendo:

- Un **constructor** de tipos `m * -> *`, que para cada tipo `a`, devuelve una mónada conteniéndolo, `m a`.
- Una **función** `return a -> m a`, que para todo elemento de tipo `a`, devuelve una mónada que lo contiene.
- Una **función** `(>>=) m a -> (a -> m b) -> m b`, que dada una mónada y una función que se aplique sobre su interior y devuelva otra mónada, devuelve la mónada resultante. Sirve como composición de funciones monádicas.

La existencia de la última función equivale a la existencia de otras dos funciones `fmap (a -> b) -> m a -> m b` y `join m (m a) -> m a`.

Nótese entonces que para ser mónada, una clase de tipos debe ser primero un funtor. Dentro de los funtores que conocemos, podemos reconocer algunas mónadas, incluyendo la mónada `Maybe` que hemos usado hasta ahora:

```haskell
-- Return de la mónada Maybe
return x = Just x

-- Bind de la mónada Maybe
(Just x) >>= k = k x
Nothing  >>= _ = Nothing

-- Return de la mónada List
return x = [x]

-- Bind de la mónada List
xs >>= f = [y | x <- xs, y <- f x]
```

Puedes empezar a leer tutoriales sobre el uso de las mónadas en Haskell en:

- A fistful of monads - Learn you a Haskell
- Understanding monads - Wikibooks

Mónada lista

En las listas, por ejemplo, tenemos como candidato para `join :: [[a]] -> [a]` la concatenación de listas, `concat`. Nuestro `return :: a -> [a]` será simplemente incluir un elemento en una lista que sólo lo contenga a él.

En esta mónada, `(>>=)` mapea una función `a -> [a]` sobre cada elemento de la lista y concatena todos los resultados:

```
> [1,2,3] >>= (replicate 3)
[1,1,1,2,2,2,3,3,3]
```

Nótese que, de la misma manera en la que podemos usar `(>>=)`, podemos usar una versión con sus parámetros cambiados de orden, `(=<<)`. Aquí usamos la mónada lista para enumerar los racionales repitiendo elementos:[2]

```haskell
import Data.List
import Data.Ratio

-- Crea los racionales con denominador n
withDenom :: Integral -> [Rational]
withDenom n = map (%n) [1..]

-- Para cada entero, crea los racionales que lo
-- tienen como denominador
rationalsDup :: [Rational]
rationalsDup = withDenom =<< [1..]

-- Evita duplicados con 'nub'
rationals :: [Rational]
rationals = nub rationalsDup
```

Mónada IO

La mónada IO surge como una solución al problema de implementar efectos secundarios (como la lectura o escritura) en un lenguaje puro y de forma extensible, sin tener que alterar el sistema de tipos y respetando el orden en el que queremos que se ejecuten.

```haskell
greeting :: IO ()
greeting = fmap ("Hola, "++) getLine >>= print
```

Pueden leerse más detalles sobre mónada IO y su implementación en:

- IO is pure - Chris Taylor
- Imperative functional programming - Simon L. Peyton Jones, Philip Wadler
- First-Class "Statements" - Justin Le

Mónada estado

En ocasiones necesitamos que nuestras funciones conserven un estado además de realizar sus operaciones. Para esos casos existe la mónada estado `State s`, que guarda un valor de estado de tipo `s`. Podemos pensar en `State s a` como `s -> (a,s)`; es decir, un elemento dentro de la mónada es una función dispuesta a tomar un estado inicial y a devolver algún elemento junto a un estado final.

Existen tutoriales sobre la mónada estado en:

- State Monad - Haskell wiki
- For a few monads more - Learn you a Haskell
- The State Monad: a tutorial for the confused - Brandon Simmons
- Three useful monads - Aditya Bhargava

Un uso de la mónada estado puede ser el guardar la semilla de una generación pseudoaleatoria de números usando generadores lineales congruenciales. En concreto, usaremos la fórmula iterativa $x_{i+1} \equiv 16807x_i \mod 2147483647$, que se expone aquí. Con este generador podremos escribir dados de un número dado de caras y llamarlos varias veces. El estado interno pasará la semilla aleatoria de un dado al siguiente:

```
import Control.Monad.State
type Seed = Int

dice :: Int -> State Seed Int
dice n = state (\s -> (s `mod` n + 1, 16807*s `mod` 2147483647))
```

Y podríamos llamarlo con la semilla `1000` de la forma siguiente; que nos devolverá por un lado el resultado de la tirada y por otro lado la nueva semilla:

```
> runState (dice 6) 1000
(5,1660)
```

Si queremos hacer varias tiradas seguidas, podemos usar `replicateM :: Int -> m a -> m [a]`, que se encarga de pasar internamente la semilla de cada tirada a la siguiente tirada:

```
> fst (runState (replicateM 100 (dice 6)) 1037)

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

Incluso podemos crear nuevos generadores aleatorios a partir de los anteriores con las operaciones usuales:

```haskell
twodices :: State Seed Int
twodices = do
    a <- dice 6
    b <- dice 6
    return (a+b)
```

Cuando lo llamemos, tomará la distribución suma de las dos distribuciones de dados:

```
> fst (runState (replicateM 100 twodices) 1032)

[8,11,8,8,5,3,7,8,9,5,2,6,7,9,8,7,5,9,3,9,10,7,7,10,
8,2,5,6,4,10,8,6,4,6,4,8,9,7,12,11,9,3,2,7,5,5,6,10,
6,6,3,11,4,7,3,6,3,7,10,4,4,11,4,10,3,5,2,8,4,10,12,
8,9,5,9,11,6,4,10,6,6,12,5,2,7,8,7,4,4,4,9,6,6,6,3,
11,11,9,7,6]
```

Mónadas en teoría de categorías

Para entender cómo funcionan las mónadas en teoría de categorías tenemos que entender dos conceptos: los *productos en una categoría monoidal* y *los endofuntores de una categoría*. La unión de ambos conceptos es lo que nos dará las mónadas como una construcción en teoría de categorías.

Categorías monoidales

Simplificando, una categoría monoidal es aquella donde, dados dos objetos A, B, tenemos un objeto *"producto tensor"* de ambos, $A \otimes B$, donde además existe un objeto identidad I cumpliendo propiedades como:

$$A \otimes B \cong B \otimes A$$

$$A \otimes (B \otimes C) \cong (A \otimes B) \otimes C$$

$$A \otimes I \cong A$$

Ejemplos de categorías monoidales

Los **conjuntos** con el producto cartesiano y el conjunto de un elemento forman ya una categoría monoidal. Puede comprobarse sobre ellos que existen los isomorfismos:

$$A \times B \cong B \times A$$

$$A \times (B \times C) \cong (A \times B) \times C$$

$$A \times \{\bullet\} \cong A$$

Pero además, podemos darles *otra* estructura de categoría monoidal, esta vez con la unión disjunta y el conjunto vacío:

$$A \sqcup B \cong B \sqcup A$$

$$A \sqcup (B \sqcup C) \cong (A \sqcup B) \sqcup C$$

$$A \sqcup \varnothing \cong A$$

En general, **todas las categorías con productos finitos son categorías monoidales** con el producto categórico y el objeto terminal como unidad. Todas las categorías con coproductos finitos son categorías monoidales con el coproducto categórico y el objeto inicial como unidad.

Otro ejemplo distinto lo forman los **espacios vectoriales** sobre un cuerpo K con el producto tensor y el propio cuerpo sirviendo como unidad; o los **grupos abelianos** con el producto tensor y $\mathbb{Z}$ siendo la unidad.

Objetos monoide

Un objeto A de una categoría monoidal es objeto monoide cuando puedo definir un morfismo desde el objeto identidad hacia él y un morfismo desde el producto tensor $A \otimes A$ hacia él. Es decir, hay un morfismo *unidad*, $I \xrightarrow{u} A$; y hay un morfismo *multiplicación*, $A \otimes A \xrightarrow{\mu} A$.

Cumpliendo ciertas propiedades similares a las que exigimos a un monoide. De hecho, un objeto monoide en la categoría de los conjuntos con el producto cartesiano es simplemente un **monoide** normal y corriente.

Categorías de endofuntores

El ejemplo que nos interesa ahora, sin embargo, es el de los **endofuntores** de una categoría. Un **funtor**, de forma simplificada, es una *aplicación entre categorías*; que lleva objetos en objetos y morfismos en morfismos, respetando además el punto de inicio y fin de cada morfismo.

Si consideramos los funtores de una categoría a sí misma, tenemos los **endofuntores** de la categoría. Y entre ellos existen transformaciones naturales que actúan como morfismos en el sentido de que se componen para dar otras transformaciones naturales. Teniendo unos objetos (los endofuntores), y unos morfismos (las transformaciones naturales), tenemos una categoría. Nótese que hemos abstraído mucho, estamos trabajando con una categoría en la que cada objeto es en sí mismo un endofuntor y cada morfismo es toda una transformación natural entre dos funtores.

Esta es además una categoría monoidal. El producto tensor de esta categoría monoidal será la composición $\circ$, y el objeto identidad el endofuntor identidad, que actúa dejando fijo cada objeto y cada morfismo.

Mónadas

Pues bien, una mónada es un objeto monoide en la categoría de los endofuntores con la composición como producto tensor. Esto quiere decir que es un endofuntor F con transformaciones naturales:

$$F \circ F \Rightarrow F$$

$$I \Rightarrow F$$

Como una transformación natural nos da un morfismo por cada objeto en el que se aplica el funtor, lo que tenemos son familias de morfismos:

$$F(F(X)) \xrightarrow{\mu_x} F(X)$$

$$X \xrightarrow{r_x} F(X)$$

Una categoría para la programación funcional

Algunos sistemas de tipos, con las funciones entre ellos `A -> B` como morfismos, forman una categoría[3]. No es el caso de Haskell, en el que, por varios motivos, sus tipos no forman una categoría[4]; pero las construcciones con inspiración en la teoría de categorías toman el nombre de sus homólogas.[5]

El primer ejemplo de esto son los **funtores**. En Haskell, un funtor se define como:

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

Es decir un funtor toma un tipo `a` (un objeto de la categoría), y nos devuelve otro tipo `f a` (otro objeto de la *misma* categoría). Por otro lado, el funtor toma un morfismo `a -> b` y nos devuelve otro morfismo `f a -> f b`. Es decir, los funtores de la programación funcional son **endofuntores en la categoría de los tipos**, siempre que conserven ciertas reglas que tienen su reflejo en las reglas de funtores en Haskell.

Monoides en la categoría de los endofuntores

Entonces, si los funtores `f :: * -> *` son endofuntores en alguna categoría, tiene sentido cuestionarse cuáles de ellos son monoides. Esto, junto con ciertas restricciones que se le imponen como leyes a las mónadas, equivale a decir que existen los morfismos dados por las transformaciones naturales que pedíamos antes, es decir, deben existir morfismos de tipos `a -> m a` y `m (m a) -> m a`:

```
return :: a -> m a
join :: m (m a) -> m a
```

Así, a cualquier funtor que tiene estos dos morfismos, además del `fmap` que tenía por ser funtor, lo llamamos **mónada**. Nótese que `>>=` puede implementarse desde `join` y viceversa, como:

```haskell
(>>=) :: m a -> (a -> m b) -> m b
(>>=) x f = join (fmap f a)

join :: m (m a) -> m a
join x = x >>= id
```

Tenemos entonces dos definiciones equivalentes de lo que es una mónada. Una desde la teoría de categorías y otra desde la teoría de tipos y los lenguajes de programación funcional. Una visión desde las mónadas como monoides en la categoría de los endofuntores desde dentro de Haskell puede verse aquí.

Y más

Además de las enunciadas en este post, existen más otros temas relevantes en relación a las mónadas, tanto en programación funcional como en teoría de categorías:

- Transformadores de mónadas, usadas para componer mónadas.
- Comónadas, la noción dual de una mónada.
- Categorías de Kleisli, cada mónada da lugar a una categoría de Kleisli.
- Funtores adjuntos, cada par de funtores adjuntos da lugar a una mónada.
- Typeclassopedia, una revisión de otras clases de tipos relevantes en Haskell y relacionadas con la teoría de categorías.

Referencias

[1] Peligros de la notación do. Do notation considered harmful

[2] Enumerando los racionales. Enumerating the rationals - J. Gibbons, D. Lester, R. Bird

[3] La correspondencia entre tipos, lógica y categorías. Computational Trinitarianism - NLab

[4] Por qué los tipos de Haskell no son una categoría. Hask is not a category - Andrej Bauer

[5] Por qué en ocasiones puede ser útil pensar en ellos como una categoría. Does it matter if Hask is (not) a category?