Mónadas en Haskell

Pablo Andrés Martínez
Eduardo García Ruiz

¿Qué son los Monad?

- Es un concepto muy abstracto propio de lenguajes funcionales.
- Sin embargo, llevan en Haskell desde sus inicios (Haskell 1.3).
- Todos hemos usado mónadas aunque no lo sepamos: *List*, *Maybe*, *IO*…
- Hay varios conceptos que tenemos que abordar antes de intentar entender los Monad.

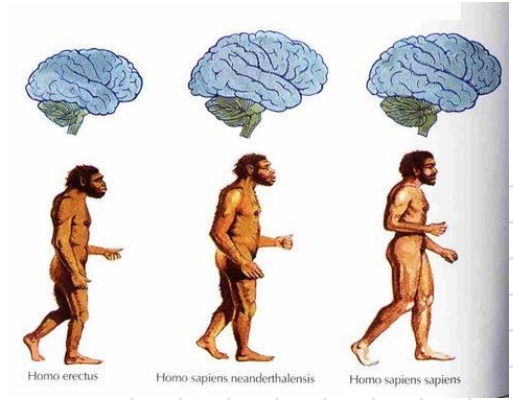
¿Contexto?¿Qué es eso?

```
data Type t = Wrap t
```

- Toda estructura de datos tiene un **contexto asociado**.
- Podemos entender una estructura de datos como un **envoltorio** que proporciona cierto **contexto** a su contenido. Un entorno, significado.
- Por ejemplo, el contexto de ***Maybe*** es que la computación puede devolver un resultado válido (*Just x)* o puede dar lugar a un error (*Nothing*).

Monad y sus superclases

- Monad es la última capa de una serie de abstracciones que van en progresión.
- El objetivo final es poder trabajar con contextos de forma **transparente**.
- Así pues, definiremos previamente dos clases con menor potencia que Monad a la hora de **manejar contexto**: *Functor* y *Applicative*
- Como ya explicaremos, éstas **son superclases** de Monad



Functor

- Queremos modificar valores situados en un contexto.
- ¿Cómo lo hacemos?
  - Functor define *fmap* para transformar el contenido envuelto en un contexto:

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

  - Leyes de Functor:

```
fmap id = id
fmap (p.q) = (fmap p).(fmap q)
```

Functor: Maybe

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

- Vemos que Functor maneja contextos con cierta transparencia …
  - *fmap* se encarga de distinguir el comportamiento de la función sobre el contexto según el estado del mismo.
  - Si intentamos mapear f sobre *Nothing*, obtenemos *Nothing*.

Applicative

- Functor presenta ciertas limitaciones a la hora de trabajar con un contexto
- Applicative es una mejora de Functor
  - Permite mapear funciones envueltas

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

  - Leyes de Applicative:

```
pure f <*> x = fmap f x
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

Applicative: Operador <*>

- (<*>) nos permite aplicar funciones N-arias envueltas
  - Es muy similar a *fmap*, pero la función a mapear se encuentra dentro del ***contexto.***
  - Es asociativo por la izquierda.
  - Evalúa los contextos a los que se aplica paso-por-paso.

```
pure g <*> f x_1 <*> f x_2 <*> ... <*> f x_N

((((pure g <*> f x_1) <*> f x_2) <*> ...) <*> f x_N)

((((f g <*> f x_1) <*> f x_2) <*> ...) <*> f x_N)

(((f (g x_1) <*> f x_2) <*> ...) <*> f x_N)

((f (g x_1 x_2) <*> ...) <*> f x_N)

f (g x_1 x_2 ... x_N)
```

Applicative: Operador <$>

- Si a la expresión:

```
pure g <*> f x_1 <*> f x_2 <*> ... <*> f x_N
```

le aplicamos la ley **pure f <*> x = fmap f x** , obtenemos la expresión:

```
fmap g x_1 <*> f x_2 <*> ... <*> f x_N
```

⇨ Para realizar este tipo de operaciones, **Haskell** nos proporciona *azúcar sintáctico* exportando la función **<$>**.

⇨ La función **<$>** no es más que la función ***fmap*** en notación infija.

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

Applicative: Maybe

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> mx = fmap f mx
```

- Contexto por defecto: ***Just***
- Como todo Applicative debe ser instancia de Functor, podemos usar *fmap* para implementar el operador **<*>**
- **(<*>)** devuelve *Nothing* si aparece *Nothing* en alguno de sus argumentos.

Monad

- Applicative no permite que el usuario indique cómo debe evolucionar el contexto.
  - Por ello, no será posible definir comportamientos del tipo “aplica `(-) <$> Just N <*> Just 1` pero devuelve *Nothing* si N ≤ 0”
  - Monad sí permite resolver este problema:

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

- Leyes:

```
return x >>= f = f x
m >>= return = m
(m >>= f) >>= g = m >>= (\y -> f y >>= g)
```

Monad: Operador bind (>>=)

- Con él, el usuario puede indicar **cómo se actualiza el contexto** de la estructura de datos.

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

  - El primer argumento es la *estructura de datos previa*.
  - El segundo argumento es la función que, **dependiendo del contenido** que tenga la *estructura de datos previa*, devuelve un nuevo valor **envuelto en cierto contexto**.
  - El operador maneja el resto del contexto de forma transparente al usuario, dando como salida una estructura de datos coherente.

Monad: Maybe

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

- Contexto por defecto: ***Just***
- Contexto en error: ***Nothing***

- Maneja el contexto de forma transparente:
  - Si el primer argumento es *Nothing*, devuelve *Nothing*
  - En caso contrario, aplica la función del usuario, pudiendo obtenerse como salida *Just* o *Nothing*.

Monad: *do notation*

- Permite trabajar con los tipos monádicos desde un punto de vista **imperativo.**
- Es implementado por Haskell de forma automática, basándose en (>>=):

```
do v <- m1          ≡   m1 >>= (\x -> case x of v -> m2
   m2                                    _ -> fail "s")
```

- Un ejemplo:

```
restaUno :: Maybe Int -> Maybe Int
restaUno m = do x <- m
                True <- return (x > 0)
                return (x-1)
```

Monad y sus superclases II

- Todo Monad es un Applicative:

```
instance Applicative m where
    pure = return
    mf <*> mx = do f <- mf
                   x <- mx
                   return (f x)
```

- Todo Applicative es un Functor:

```
instance Functor m where
    fmap f mx = (pure f) <*> mx
```

- Por tanto, Functor es superclase de Applicative y éste de Monad.

Monad: List

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

- Contexto por defecto: ***Lista de un elemento***
- Contexto en error: ***Lista vacía.***

- Contexto propuesto por List: ***no-determinismo***.
- ***do notation*** equivalente a ***List comprehension***.

```
do a<-[2,3,7]
   b<-[11,13,19]
   return (a*b)
```

⟺

```
[ a*b | a<-[2,3,7], b<-[11,13,19]]
```

Monad: State Monad

- Envuelve una computación:

```
data MS s a = C (s -> (a,s))
```

- Su implementación es la siguiente:

```
instance Monad (MS s) where
    (>>=) (C c1) fc2 = C (\xs -> let
                                    (r,xs') = c1 xs
                                    (C c2) = fc2 r
                                in c2 xs')
    return k = C (\xs -> (k,xs))
```

- El operador *bind* nos permitirá **secuenciar computaciones**, actualizando el estado de forma transparente.

- Una función ***run*** desenvolverá la función y la ejecutará.

Stack con State Monad

- El estado interno de un Stack es la colección de elementos ordenados.

```
type Stack t a = MS [t] a
```

- Debemos definir algunas funciones para darle utilidad al tipo:

  - ***pop***: Extrae el elemento en el *top* del Stack
  - ***push x***: Introduce un elemento en el Stack
  - ***stackLength***: Devuelve la longitud del Stack

- Un ejemplo de su aplicación: *operar expresiones en Not. Polaca*

¿Cuándo utilizar Monads?

- En caso de querer **definir tipos** de datos que **manejen contextos mutables**.

- En caso de querer **utilizar** herramientas que manejen el **contexto** del problema de una forma **transparente**:

  - **Maybe**: Estado de error de la computación ejecutada.
  - **List**: No determinismo.
  - **IO**: Interfaz de flujo de entrada/salida con el usuario.
  - **Either**: Cuando puedan devolverse dos mensajes distintos.
  - **State Monad**: Cuando necesitemos manejar un estado.

¡Qué monada de While!

- El estado interno de nuestro While será el tipo **State**

```
type While a = MS State a
```

- Definimos funciones para trabajar con nuestro **State** interno:
  - **assW:** Modifica el estado interno añadiendo una nueva relación Var -> Z o modificando una ya existente.
  - **skipW:** No modifica el estado.
  - **compW:** No modifica el estado **explícitamente**. Establece un orden entre las operaciones.
  - **ifW:** Implementa la semántica del if.
  - **whileW:** Implementa la semántica del while.

Referencias

- **Learn You a Haskell for Great Good!**
  - http://learnyouahaskell.com/chapters
- **All About Monads**
  - https://wiki.haskell.org/All_About_Monads
- **Tutorials of Monad**
  - https://wiki.haskell.org/Tutorials#Using_monads
- **Ejercicios intermedios de Haskell**
  - http://blog.tmorris.net/posts/20-intermediate-haskell-exercises
- **Wikipedia**
  - https://es.wikipedia.org/wiki/M%C3%B3nada_(programaci%C3%B3n_funcional)