

Programación Funcional Avanzada

Monads – Introducción

Ernesto Hernández-Novich

<emhn@usb.ve>

Universidad “Simón Bolívar”

Copyright ©2010-2016

La historia hasta ahora...

- Functor – $fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$
 - Valores en un contexto de cómputo.
 - Manipulables con funciones externas manteniendo el contexto.
- Applicative – $<*> :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$
 - Valores y funciones en un contexto de cómputo.
 - Combinables en secuencia, generalmente como aplicaciones parciales que conducen a una aplicación final y resultado.



La historia hasta ahora. . .

- Functor – `fmap :: (a -> b) -> f a -> f b`
 - Valores en un contexto de cómputo.
 - Manipulables con funciones externas manteniendo el contexto.
- Applicative – `<*> :: f (a -> b) -> f a -> f b`
 - Valores y funciones en un contexto de cómputo.
 - Combinables en secuencia, generalmente como aplicaciones parciales que conducen a una aplicación final y resultado.
- Ninguno de los dos puede modelar

```
oddEnough act = do
  v <- act
  if odd v
    then fail "too odd!"
    else return v
```

porque una secuencia de `fmap` o `<*>` es *incapaz* de examinar resultados intermedios y tomar decisiones.

Monads

¿Qué son y para qué sirven?

- Originalmente de la Teoría de Categorías
 - Eugenio Moggi lo aplicó a semántica de lenguajes.
 - Philip Wadler mostró como usarlas explícitamente en Haskell.
- Definición(es?)
 - Estructurar cálculos en términos de valores intermedios que influyen en la secuencia de cálculos que los utilizan.
 - Bloques “combinables” (*composable*) para secuenciar cálculos.
 - Estrategia para construir cálculos “complejos” partiendo de “simples”.
- Naturaleza abstracta – múltiples aplicaciones.
 - Modularidad – separar estrategia del cálculo específico.
 - Flexibilidad – centralizar la estrategia de cálculo.
 - Aislamiento – separar procesamiento puro de impuro.



La clase Monad

... salgamos de esto

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

  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a
```

- m es el constructor de tipos Monad cuyo contenido es arbitrario.
- Se *inyectan* valores en el Monad con `return` – nombre *muy* desafortunado, pero ya es tarde para cambiarlo.
- La secuencia de cálculos opera *dentro* del contenedor para producir un nuevo contenedor, posiblemente con un contenido diferente.



¿De dónde te conozco?

- El inyector para Monad

```
return :: a -> m a
```



¿De dónde te conozco?

- El inyector para Monad

```
return :: a -> m a
```

- El inyector para Applicative

```
pure :: a -> f a
```



¿De dónde te conozco?

- El inyector para Monad

```
return :: a -> m a
```

- El inyector para Applicative

```
pure :: a -> f a
```

- ¡Se parecen igualito!
 - Porque son exactamente el mismo comportamiento – llevar un valor puro a un contexto de cómputo.
 - Monad es un Applicative en esteroides.
- Monad llegó a Haskell mucho antes que Applicative
 - Matemáticamente, todo Monad es un Applicative.
 - La jerarquía de clases de tipos lo refleja a partir de GHC 7.8 – no influye ni impide nuestra habilidad de usarlas.



¿Y qué mejoras obtengo?

- Todo Monad es un Functor – seguimos teniendo fmap.

```
fmap :: (a -> b) -> m a -> -> m b
```

(cambié f por m)



¿Y qué mejoras obtengo?

- Todo Monad es un Functor – seguimos teniendo `fmap`.

```
fmap :: (a -> b) -> m a -> -> m b
```

(cambié `f` por `m`)

- Todo Monad es un Applicative – seguimos teniendo `<*>`.

```
(<*>) :: m (a -> b) -> m a -> m b
```

(cambié `f` por `m`)



¿Y qué mejoras obtengo?

- Todo Monad es un Functor – seguimos teniendo `fmap`.

```
fmap :: (a -> b) -> m a -> -> m b
```

(cambié `f` por `m`)

- Todo Monad es un Applicative – seguimos teniendo `<*>`.

```
(<*>) :: m (a -> b) -> m a -> m b
```

(cambié `f` por `m`)

- ¿Cuál es la mejora de `>>=` (*bind*)?

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

- Levantar el valor del contexto.
- Procesarlo con una función que produce un nuevo contexto.

`j>>=` permite manipular los valores intermedios!



Las leyes Monad

... salgamos de esto

Un Monad “bien formado” debe respetar las siguientes leyes



Las leyes Monad

... salgamos de esto

Un Monad “bien formado” debe respetar las siguientes leyes

- 1 `return` es la identidad izquierda con respecto a `>>=`

```
(return x) >>= f == f x
```



Las leyes Monad

... salgamos de esto

Un Monad “bien formado” debe respetar las siguientes leyes

- 1 return es la identidad izquierda con respecto a $>>=$

```
(return x) >>= f == f x
```

- 2 return es la identidad derecha con respecto a $>>=$

```
m >>= return == m
```



Las leyes Monad

... salgamos de esto

Un Monad “bien formado” debe respetar las siguientes leyes

- ❶ return es la identidad izquierda con respecto a $\gg=$

```
(return x) >>= f == f x
```

- ❷ return es la identidad derecha con respecto a $\gg=$

```
m >>= return == m
```

- ❸ $\gg=$ es asociativo

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



Las leyes Monad

... salgamos de esto

Un Monad “bien formado” debe respetar las siguientes leyes

- 1 return es la identidad izquierda con respecto a $\gg=$

```
(return x) >>= f == f x
```

- 2 return es la identidad derecha con respecto a $\gg=$

```
m >>= return == m
```

- 3 $\gg=$ es asociativo

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

El compilador **no** puede verificarlo.
Es responsabilidad del programador.

Faltaron dos combinadores Monad

... salgamos de esto

- `>>` permite combinar operaciones ignorando el resultado intermedio

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

- La secuencia ignora el valor intermedio.
- Util cuando sólo interesan los efectos de borde.



Faltaron dos combinadores Monad

... salgamos de esto

- `>>` permite combinar operaciones ignorando el resultado intermedio

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

- La secuencia ignora el valor intermedio.
 - Util cuando sólo interesan los efectos de borde.
- `fail` permite indicar un fallo en el cómputo

```
fail s = error s
```

- Permite terminar “dramáticamente” la secuencia.
- No existe en la definición matemática – es necesaria (“rancho”) para la operación del lenguaje.

Monad provee estas implantaciones por omisión, el programador puede proveer una más eficiente si conviene.



¿Cómo “escapo” de un Monad?

... salgamos de esto

- Monad no define un mecanismo para “sacar” valores del contexto – esto es intencional.
- Es posible construir Monads que proveen funciones adicionales para extraer todo o parte de los contenidos según sea necesario – es un **Monad transparente**.
- Es posible construir Monads sin vía de escape para permitir la interacción con el exterior sin romper las propiedades funcionales del resto del programa – es un **Monad opaco**.

Está fue la observación de Wadler que
dió lugar al Monad IO



Monad Maybe

Maybe the simplest Monad – (pun intended)

```
type Nombre      = String
type Telefono    = String
type Direccion  = String
data Empresa     = Malestar | Bobilnet | Chimbitel
```

Tenemos sendos `Data.Map` con las relaciones

- De Nombre a Telefono
- De Telefono a Empresa
- De Empresa a Direccion

¿Cómo escribimos `direccionCobro`?



Monad Maybe

Simple, pero feo por no usar Monads

```
import qualified Data.Map as M

direccionCobro :: Nombre
               -> M.Map Nombre Telefono
               -> M.Map Telefono Empresa
               -> M.Map Empresa Direccion
               -> Maybe Direccion

direccionCobro p mt me md =
  case M.lookup p mt of
    Nothing -> Nothing
    Just t   ->
      case M.lookup t me of
        Nothing -> Nothing
        Just e  -> M.lookup e md
```



Monad Maybe

Definición del Monad Maybe

```
instance Monad Maybe where
  -- Inject
  return x = Just x

  -- Bind
  Nothing >>= _ = Nothing
  Just x   >>= k = k x

  Nothing >> _ = Nothing
  Just _   >> k = k

  fail _ = Nothing
```

Maybe – cómputo que puede producir un resultado o fallar.



Monad Maybe

Brutalmente compacto ...

- Notamos la firma de la función `lookup` en `Data.Map` ...

```
lookup :: (Ord k, Monad m) => k -> Map k a -> m a
```

... que sería perfecta si los dos argumentos estuvieran al revés



Monad Maybe

Brutalmente compacto ...

- Notamos la firma de la función `lookup` en `Data.Map` ...

```
lookup :: (Ord k, Monad m) => k -> Map k a -> m a
```

... que sería perfecta si los dos argumentos estuvieran al revés

- ¡Aprovechamos la función `flip` estándar!

```
flip :: (a -> b -> c) -> b -> a -> c  
flip f x y = f y x
```



Monad Maybe

Brutalmente compacto ...

- Notamos la firma de la función `lookup` en `Data.Map` ...

```
lookup :: (Ord k, Monad m) => k -> Map k a -> m a
```

... que sería perfecta si los dos argumentos estuvieran al revés

- ¡Aprovechamos la función `flip` estándar!

```
flip :: (a -> b -> c) -> b -> a -> c  
flip f x y = f y x
```

- Y la solución cabe en esta lámina.

```
import qualified Data.Map as M  
direccionCobro p mt me md =  
  look mt p >>= look me >>= look md  
  where look = flip M.lookup
```



Notación do para Monads

Syntactic sugar for the win!

- Manera alternativa y más clara, de escribir código monádico.
- Traducción directa hacia los combinadores monádicos.
 - $m \gg= f$ es lo mismo que $m \gg= (\backslash x \rightarrow f\ x)$
 - $m \gg f$ es lo mismo que $m \gg= (\backslash_ \rightarrow f)$



Notación do para Monads

Syntactic sugar for the win!

- Manera alternativa y más clara, de escribir código monádico.
- Traducción directa hacia los combinadores monádicos.
 - `m >>= f` es lo mismo que `m >>= (\x -> f x)`
 - `m >> f` es lo mismo que `m >>= (_ -> f)`
- Basta “voltear las flechas”
 - `m >>= (\x -> f x)` se convierte en

```
do x <- m
   f x
```

- `m >>= (_ -> f)` se convierte en

```
do m
   f
```



Notación do para Monads

Reescribamos nuestra solución

Con notación do – *Syntactic Sugar Rush*

```
direccionCobro p mt me md = do
  n <- M.lookup p mt
  e <- M.lookup n me
  M.lookup e md
```



Notación do para Monads

Reescribamos nuestra solución

Con notación do – *Syntactic Sugar Rush*

```
direccionCobro p mt me md = do
  n <- M.lookup p mt
  e <- M.lookup n me
  M.lookup e md
```

Que en realidad se traduce como

```
direccionCobro p mt me md =
  M.lookup p mt >>= (\n ->
  M.lookup n me >>= (\e ->
  M.lookup e md))
```

(Los paréntesis no son necesarios)



Escapando de Maybe

- Usar *pattern matching* para determinar si un valor es Just o Nothing.
- Usar las funciones de `Data.Maybe` – “correr” un Monad Maybe.

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

```
> maybe 69 (2*) Nothing
```

```
69
```

```
> maybe 69 (2*) (Just 21)
```

```
42
```

El último parámetro podría ser un cómputo monádico Maybe arbitrario

- La función *procesa* el cómputo y *extrae* el contenido del Monad.

Monads para todo público

- Identity – Cómputo trivial
- Maybe – Cómputos que producen uno o ningún resultado
- List – Cómputos que producen múltiples resultados (no-determinismo).
- Error – Cómputos que pueden fallar (excepciones)
- IO – Cómputos que realizan operaciones de I/O
- State – Cómputos que mantienen estado (transiciones)
- Reader – Cómputos que leen de un ambiente constante
- Writer – Cómputos que escriben resultados
- Cont – Computos que pueden ser interrumpidos y reiniciados (continuaciones)



Monad Identity

Si, es trivial ...

El cómputo se aplica al valor contenido ... duh!

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
  -- Inject
  return x          = Identity x
  -- Bind
  (Identity x) >>= f = f x
```

- No representa *ninguna* estrategia de cómputo.
- Simplemente aplica la función a su entrada sin modificarla.
- Notación de registro para tener la función de extracción de una.
- Permiten **transformar** Monads – lo estudiaremos más adelante.



Escapando trivialmente

Sólo para afianzar conocimientos

```
trivial n m = do
  x <- m
  y <- Identity (n*x)
  z <- Identity (even y)
  return z

> runIdentity $ trivial 21 (Identity 2)
True
```



Escapando trivialmente

Sólo para afianzar conocimientos

```
trivial n m = do
  x <- m
  y <- Identity (n*x)
  z <- Identity (even y)
  return z
```

```
> runIdentity $ trivial 21 (Identity 2)
True
```

...sin “azúcar”

```
runIdentity $ (\n m -> m >>=
               \x -> Identity (n*x) >>=
               \y -> Identity (even y)) 21 (Identity 2)
```



Monad List

No-determinismo

```
instance Monad [] where
  -- Inject
  return x = [x]
  -- Bind
  m >>= f  = concatMap f m

  fail    s = []
```

- Estrategia de combinar una cadena de cómputos no-determinísticos.
- Aplica la función a todos los valores posibles en cada paso, recopilando los resultados.
- Útiles para construir cómputos donde hay ambigüedad.



Deduciendo el bind para Monad List

¿Cuál es el tipo de >>=?

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```



Deduciendo el bind para Monad List

¿Cuál es el tipo de >>=?

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

¿Cuál es el tipo de map?

```
map :: (a -> b) -> [a] -> [b]
```

... si tan sólo map fuese “al revés”



Deduciendo el bind para Monad List

¿Cuál es el tipo de $>>=$?

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

¿Cuál es el tipo de map?

```
map :: (a -> b) -> [a] -> [b]
```

... si tan sólo map fuese “al revés”

```
flip map :: [a] -> (a -> b) -> [b]
```

Claro, pero el b no cuadra con el resto

Deduciendo el bind para Monad List

¿Cuál es el tipo de `>>=`?

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

¿Cuál es el tipo de `map`?

```
map :: (a -> b) -> [a] -> [b]
```

...si tan sólo `map` fuese “al revés”

```
flip map :: [a] -> (a -> b) -> [b]
```

Claro, pero el `b` no cuadra con el resto

```
concat :: [[a]] -> [a]
```

```
\xs f -> concat (map f xs) :: [a] -> (a -> [b]) -> [b]
```

Y todo encaja perfectamente ...



¿Cómo usar el Monad List?

¿Recuerdan las listas por comprensión?

```
cartesiano xs ys = [ (x,y) | x <- xs, y <- ys ]
```



¿Cómo usar el Monad List?

¿Recuerdan las listas por comprensión?

```
cartesiano xs ys = [ (x,y) | x <- xs, y <- ys ]
```

Eso es equivalente a

```
cartesiano xs ys = do
  x <- xs
  y <- ys
  return (x,y)
```



¿Cómo usar el Monad List?

¿Recuerdan las listas por comprensión?

```
cartesiano xs ys = [ (x,y) | x <- xs, y <- ys ]
```

Eso es equivalente a

```
cartesiano xs ys = do
  x <- xs
  y <- ys
  return (x,y)
```

... y sin “azúcar” debe ser obvio lo que pasa

```
cartesiano xs ys = do
  xs >>= \x ->
  ys >>= \y ->
  return (x,y)
```



¿Cómo usar el Monad List?

Sustituyamos la definición de `>>=` para aclarar más ...

```
cartesiano xs ys = do
  concat (map (\x ->
              concat (map (\y ->
                          return (x,y))
                          ys))
          xs)

let ns = [1,2,3]
    bs = [True,False]
in cartesiano ns bs
```

Ejecutarlo a mano ayuda a comprender.



Soluciones por “fuerza bruta”

Encontrar x e y tales que $x \times y = n$

```
guarded :: Bool -> [a] -> [a]
guarded True  xs = xs
guarded False _  = []

producto :: Int -> [(Int,Int)]
producto n = do
  x <- [1..n]
  y <- [x..n]
  guarded (x*y == n) $
    return (x,y)

> producto 8
[(1,8),(2,4)]
> producto 100
[(1,100),(2,50),(4,25),(5,20),(10,10)]
```

Moviendo un caballo por el tablero

Presumo que conocen algo de ajedrez

```
type KnightPos = (Int,Int)

moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = do
  (c',r') <- [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
             ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
             ]
  guard (c' `elem` [1..8] && r' `elem` [1..8])
  return (c',r')
```

- Dada un posición inicial calcula todos los posibles destinos.
- guard – según el predicado, retorna la lista resultante o [].



Moviendo un caballo por el tablero

Las secuencias se hacen triviales

```
> return (1,1) >>= moveKnight
[(3,2),(2,3)]
> return (1,1) >>= moveKnight >>= moveKnight
[(5,1),(5,3),(1,1),(1,3),(4,4),(2,4),
 (4,2),(4,4),(3,1),(3,5),(1,1),(1,5)]
```



Funciones no-monádicas y funciones monádicas

- Tengo un función pura y quiero aplicarla al contenido de un Monad

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f = \m -> m >>= return . f
```

```
> liftM (2*) (Just 21)
Just 42
> liftM (replicate 5) ['a']
["aaaaa"]
```

¿Pueden ver que liftM no es más que fmap?



Funciones no-monádicas y funciones monádicas

- Tengo un función pura y quiero aplicarla al contenido de un Monad

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f = \m -> m >>= return . f
```

```
> liftM (2*) (Just 21)
Just 42
> liftM (replicate 5) ['a']
["aaaaa"]
```

¿Pueden ver que liftM no es más que fmap?

- ¿Y si mi función pura es de más de un argumento?

```
liftM2 :: (Monad m) => (a -> b -> c) ->
          m a -> m b -> m c
```

...hasta liftM5

Cruzando el puente

¡mi función es n-aria para el n que se me antoje!

¡Métela en un Monad!

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap = liftM2 ($)
```

Gracias a la curryficación ...

```
liftMn f x1 x2 ... xn
```

...se convierte en

```
return f 'ap' x1 'ap' x2 'ap' ... 'ap' xn
```

Cruzando el puente

¡mi función es n-aria para el n que se me antoje!

¡Métela en un Monad!

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap = liftM2 ($)
```

Gracias a la curryficación ...

```
liftMn f x1 x2 ... xn
```

...se convierte en

```
return f 'ap' x1 'ap' x2 'ap' ... 'ap' xn
```

Así se pueden hacer cosas como

```
> [(+1), (+2)] 'ap' [1,2,3]
[2,3,4,3,4,5]
> Just (*2) 'ap' Just 21
Just 42
```

En el otro sentido

- Aplicación de funciones se hace de derecha a izquierda, pero la secuenciación monádica de izquierda a derecha. ¿Cómo combinar ambas cosas sin “chocar”?

```
f = << x = x >> = f
```



En el otro sentido

- Aplicación de funciones se hace de derecha a izquierda, pero la secuenciación monádica de izquierda a derecha. ¿Cómo combinar ambas cosas sin “chocar”?

```
f = << x = x >> = f
```

- Ahora podemos construir una secuencia de funciones puras que se aplican de derecha a izquierda, e inyectar el resultado en una secuencia monádica desde la derecha

```
> length . replicate 2 = << [1,2,3]
6
```

En el otro sentido

- Aplicación de funciones se hace de derecha a izquierda, pero la secuenciación monádica de izquierda a derecha. ¿Cómo combinar ambas cosas sin “chocar”?

```
f = << x = x >>= f
```

- Ahora podemos construir una secuencia de funciones puras que se aplican de derecha a izquierda, e inyectar el resultado en una secuencia monádica desde la derecha

```
> length . replicate 2 = << [1,2,3]
6
```

- Y así lograr el famoso *one-liner*

```
print . length . lines = << getContents
```



Quiero saber más...

- Documentación de `Control.Monad`
- A tour of the Haskell Monad functions

