Programación Funcional Avanzada

Monads – Writer, State, IO y combinadores

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Copyright ©2010-2016

UNIVERSIDAD SIMÓN BOLÍVAR

Monad `Writer`

Ambiente de cómputo con acumulador

- Los `Monad` que conocemos hasta ahora nos permiten construir un entorno de cómputo para secuenciar operaciones.
- ¿Y si quiero *acumular* información a medida que progreso?
  - No se trata de *múltiples* resultados – además del valor arbitrario objetivo del cálculo, tenemos otro valor colección de resultados o información intermedia relevante.
  - Esa colección *sólo* puede crecer en tamaño.
  - Como la bitácora de eventos de una aplicación.

UNIVERSIDAD SIMÓN BOLÍVAR

Monad `Writer`

Ambiente de cómputo con acumulador

- Los `Monad` que conocemos hasta ahora nos permiten construir un entorno de cómputo para secuenciar operaciones.
- ¿Y si quiero *acumular* información a medida que progreso?
  - No se trata de *múltiples* resultados – además del valor arbitrario objetivo del cálculo, tenemos otro valor colección de resultados o información intermedia relevante.
  - Esa colección *sólo* puede crecer en tamaño.
  - Como la bitácora de eventos de una aplicación.
- Con esas premisas, el Monad debería
  - Operar sobre un valor y resultados acumulados – `(a,[c])`.
  - Para transformar el valor (computar) – `(a -> (b,[c]))`.
  - Preservar el resultado y los acumulados – `(b,[c])`.

UNIVERSIDAD SIMÓN BOLÍVAR

Monad `Writer`

Ambiente de cómputo con acumulador

- Los `Monad` que conocemos hasta ahora nos permiten construir un entorno de cómputo para secuenciar operaciones.
- ¿Y si quiero *acumular* información a medida que progreso?
  - No se trata de *múltiples* resultados – además del valor arbitrario objetivo del cálculo, tenemos otro valor colección de resultados o información intermedia relevante.
  - Esa colección *sólo* puede crecer en tamaño.
  - Como la bitácora de eventos de una aplicación.
- Con esas premisas, el Monad debería
  - Operar sobre un valor y resultados acumulados – `(a,[c])`.
  - Para transformar el valor (computar) – `(a -> (b,[c]))`.
  - Preservar el resultado y los acumulados – `(b,[c])`.

¡Generalizable si usamos `Monoid` en lugar de `[a]`!

Monad `Writer`

Ambiente de cómputo con acumulador

- Comencemos por definir un tipo para el cálculo

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

- El `Monad` es `Writer w` – polimórfico en `a`
  - `a` es el tipo de datos para el cómputo en secuencia.
  - `w` es el tipo de datos acumulador de información.
- El envoltorio de costumbre con su función de extracción.

UNIVERSIDAD SIMÓN BOLÍVAR

Monad `Writer`

Ambiente de cómputo con acumulador

- Instancia lo más general posible

```
instance (Monoid w) => Monad (Writer w) where
```

Monad `Writer`

Ambiente de cómputo con acumulador

- Instancia lo más general posible

```
instance (Monoid w) => Monad (Writer w) where
```

- Inyectar un valor – tupla con el valor y acumulador vacío

```
return :: a -> Writer w a
return x = Writer (x, mempty)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Monad `Writer`

Ambiente de cómputo con acumulador

- Instancia lo más general posible

```
instance (Monoid w) => Monad (Writer w) where
```

- Inyectar un valor – tupla con el valor y acumulador vacío

```
return :: a -> Writer w a
return x = Writer (x, mempty)
```

- Combinar un cómputo – calcular el nuevo valor y acumular

```
(>>=) :: Writer w a -> (a -> Writer w b)
      -> Writer w b
(Writer (x,v)) >>= f =
      let (Writer (y, v')) = f x
      in Writer (y, v `mappend` v')
```

UNIVERSIDAD SIMÓN BOLÍVAR

Monad Writer

¿Cómo aprovecharlo?

```
import qualified Data.Sequence as DS
import Control.Monad.Writer

logIt :: Int -> Writer (DS.Seq String) Int
logIt x = writer (x, DS.singleton $ "Veo " ++ show x)

multWithLog :: Writer (DS.Seq String) Int
multWithLog = do
  a <- logIt 21
  b <- logIt 2
  tell $ DS.singleton "Multiplicando."
  return $ a*b

> runWriter multWithLog
(42,fromList ["Veo 21","Veo 2","Multiplicando."])
```

- `tell` permite acumular un mensaje destruyendo el valor

UNIVERSIDAD SIMÓN BOLÍVAR

Hay más de una forma de escribir

- `Control.Monad.Writer.Lazy` provee una versión perezosa – `Control.Monad.Writer.Strict` provee una versión estricta.
- Enfasis en manipular la bitácora

```
writer :: MonadWriter w m => (a, w) -> m a
tell   :: MonadWriter w m => w -> m ()
listen :: MonadWriter w m => m a -> m (a,w)
censor :: MonadWriter w m => (w -> w) -> m a-> m a
```

UNIVERSIDAD SIMÓN BOLÍVAR

Estado mutable

¿Y esto no era programación funcional?

- Programación Funcional – Transparencia Referencial
  - No se puede modificar una estructura "en sitio".
  - Las estructuras deben reconstruirse para representar los cambios, descartando la original cuando ya no es necesaria.

UNIVERSIDAD SIMÓN BOLÍVAR

Estado mutable

¿Y esto no era programación funcional?

- Programación Funcional – Transparencia Referencial
  - No se puede modificar una estructura "en sitio".
  - Las estructuras deben reconstruirse para representar los cambios, descartando la original cuando ya no es necesaria.
- Existen cómputos basados en un **estado mutable**
  - Hay un estado "inicial".
  - En cada paso de cómputo se modifica el estado, posiblemente generando un resultado útil adicional.
  - Al terminar el cómputo puede interesar el resultado final, el estado final o ambos.

Eso parece una secuenciación de cómputos...

UNIVERSIDAD SIMÓN BOLÍVAR

Estado Mutable

Una representación

¿Qué queremos hacer con un estado?

- Establecer un estado inicial.
- Inspeccionar el estado actual.
- Producir un nuevo resultado y un nuevo estado.

Estado Mutable

Una representación

¿Qué queremos hacer con un estado?

- Establecer un estado inicial.
- Inspeccionar el estado actual.
- Producir un nuevo resultado y un nuevo estado.

Nuestro modelo

- El estado puede ser cualquier tipo de datos – `s`
- El resultado puede ser cualquier tipo de datos – `a`
- Tomar un estado para producir un resultado y un nuevo estado, se modela naturalmente como una función **transformadora de estado**

```
s -> (a,s)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Deduciendo el `State` Monad

Voy a hacer "trampa" porque puedo

- Comencemos por definir un tipo para el transformador

```
type Estado s a = s -> (a,s)
```

- Nuesto monad contendrá una **función** transformadora.
  - `s` es cualquier tipo de datos que modele el estado.
  - `a` es cualquier tipo de datos que modele resultados.

El Monad envuelve una *función* –
esa es la clave para comprender su funcionamiento.

UNIVERSIDAD SIMÓN BOLÍVAR

Deduciendo el `State Monad`

Proveer las funciones mínimas

- Inyectar un valor – función que preserve el estado, incluyendo el valor

```
returnE :: a -> Estado s a
returnE a = \s -> (a,s)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Deduciendo el State Monad

Proveer las funciones mínimas

- Inyectar un valor – función que preserve el estado, incluyendo el valor

```
returnE :: a -> Estado s a
returnE a = \s -> (a,s)
```

- Combinar un cómputo – aplicarlo y producir el nuevo estado

```
bindE :: (Estado s a)
      -> (a -> Estado s b)
      -> (Estado s b)
bindE m k = \s -> let (a, s') = m s
                  in (k a) s'
```

Si sustituimos `Estado s a` por su tipo,
puede resultar más fácil de comprender ...

UNIVERSIDAD SIMÓN BOLÍVAR

Sustituyendo `Estado s a`

```
bindE :: (s -> (a,s))
      -> (a -> s -> (b,s))
      -> (s -> (b,s))
bindE cambiar crearCambio estadoActual =
  let (resultado, estadoNuevo) = cambiar estadoActual
  in  (crearCambio resultado) estadoNuevo
```

`Estado s a` no es más que una *función*

¿Cómo interactúo con el estado?

Como siempre, las definiciones de `return` y `bind` modelan el método de cómputo, escondiendo el procesamiento. Necesitamos funciones auxiliares.

- Obtener el estado actual . . .

```
getSt :: Estado s a
getSt = \s -> (s,s)
```

. . . es *retornarlo* como si fuera el resultado.

¿Cómo interactúo con el estado?

Como siempre, las definiciones de `return` y `bind` modelan el método de cómputo, escondiendo el procesamiento. Necesitamos funciones auxiliares.

- Obtener el estado actual ...

```
getSt :: Estado s a
getSt = \s -> (s,s)
```

...es *retornarlo* como si fuera el resultado.

- Inyectar un estado particular ...

```
putSt :: s -> Estado s a
putSt s = \_ -> ((),s)
```

...es *ignorar* el actual y forzar el nuevo.

Hasta aquí llegó mi "trampa"

UNIVERSIDAD SIMÓN BOLÍVAR

Monad State

Will the real State Monad please stand up!

Tenemos que usar un tipo concreto

- Nos obliga a usar un constructor . . .
- . . . con los consecuentes empaquete y desempaquete
- Más una función de escape

```
newtype State s a = State {
  runState :: s -> (a,s)
}

instance Monad (State s) where
  return a          = State $ \s -> (a,s)
  (State x) >>= f = State $ \s -> let (v,s') = x s
                                  in runState (f v) s'
```

UNIVERSIDAD SIMÓN BOLÍVAR

Monad State

Con toda la formalidad del caso

Las funciones auxiliares están en una clase diferente

```
class MonadState m s | m -> s where
    get :: m s
    put :: s -> m ()

instance MonadState (State s) s where
    get   = State $ \s -> (s,s)
    put s = State $ \_ -> ((),s)
```

La notación `| m -> s` se lee "donde `m` determina a `s`" y se conoce como **dependencia funcional** (*fundep*).

- Ayudan al sistema de tipos a resolver ambigüedades.
- Hablaremos de ellos más adelante.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad `State`

Números pseudo-aleatorios (PRNG)

- Genera una secuencia **finita** de números que tienen propiedades similares a las de números aleatorios verdaderos.
- La secuencia
  - Basada en dos valores fijos – **multiplicador** (primo) y **módulo**.
  - Parte con un valor suministrado por el usuario – la **semilla**.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad `State`

Números pseudo-aleatorios (PRNG)

- Genera una secuencia **finita** de números que tienen propiedades similares a las de números aleatorios verdaderos.
- La secuencia
  - Basada en dos valores fijos – **multiplicador** (primo) y **módulo**.
  - Parte con un valor suministrado por el usuario – la **semilla**.
- Algebra de congruencias – usar la semilla para producir el siguiente número de la secuencia, y la nueva semilla.

$$semilla_{n+1} = (semilla_n \times multiplicador) \equiv modulo$$

$$azar_{n+1} = \frac{semilla_{n+1}}{modulo}$$

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad State

Números pseudo-aleatorios – El estado

- Definiremos el estado representando el cambio en la semilla

```
newtype PRNG a = PRNG (Integer -> (a, Integer))
```

Usando Monad State

Números pseudo-aleatorios – El estado

- Definiremos el estado representando el cambio en la semilla

```
newtype PRNG a = PRNG (Integer -> (a, Integer))
```

- Necesitamos una función para avanzar la secuencia

```
rng :: Integer -> (Float,Integer)
rng actual = ( fromInteger (nueva) /
               fromInteger(modulo),
                nueva )
  where
    nueva          = (actual * multiplicador)
                     `mod` modulo
    multiplicador = 1812433253
    modulo         = 2^32
```

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad State

Números pseudo-aleatorios – La plomería

- La instancia de Monad

```
instance Monad PRNG where
  return x = PRNG $ \semilla -> (x, semilla)

  (PRNG g0) >>= f = PRNG $
     \semilla -> let (y, semilla') = g0 semilla
                     (PRNG g1) = f y
                 in g1 semilla'
```

Usando Monad State

Números pseudo-aleatorios – La plomería

- La instancia de Monad

```
instance Monad PRNG where
  return x = PRNG $ \semilla -> (x, semilla)

  (PRNG g0) >>= f = PRNG $
     \semilla -> let (y, semilla') = g0 semilla
                     (PRNG g1) = f y
                 in g1 semilla'
```

- Funciones para comodidad

```
initialSeed          = 3121281023
random               = PRNG rng
generate (PRNG f) = (fst.f) initialSeed
```

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad State

Números pseudo-aleatorios – Uso

```
pareja = do
  a <- random
  b <- random
  return (a,b)

lista n = do
  x  <- random
  xs <- lista $ n-1
  return $ if (n == 0) then [] else (x:xs)

> generate pareja
(0.628972,0.3711084)
> generate $ lista 5
[0.628972,0.3711084,0.18037394,0.95906913,0.55127066]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Pero se puede hacer con menos código

`Control.Monad.State` es tu amigo

```
import Control.Monad.State

random :: State Integer Float
random = do
  actual <- get
  let (azar,nueva) = rng actual
  put nueva
  return azar

initialState = 3121281023
generate rs  = evalState rs initialState
```

- No hace falta el `newtype PRNG` ni escribir la instancia.
- Basta escribir firmas correctas para los transformadores.
- Todo lo demás funciona exactamente igual.

UNIVERSIDAD SIMÓN BOLÍVAR

Y puede ahorrar *mucho* tiempo

Si sabes lo que estás haciendo...

- El secreto es identificar los tipos para estado y valor – supongamos una pila simulada con una lista
  - El estado – la pila como un todo.
  - El valor – lo que se obtiene al hacer pop.
- Esto es todo lo que hace falta...

```
type Stack a = [a]

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

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

¡Pilas monádicas genéricas!

Pilas monádicas genéricas...

```
> runState (push 3 >> push 2 >> pop >> push 1) []
((),[1,3])
> runState (push 'o' >>
            pop >>= \x -> push x >> push x >>
            push 'f') []
((),"foo")
> runState (push "foo" >> push "bar" >>
            put ["wtf?"]) []
((),["wtf?"])
> runState (push "foo" >> push "bar" >>
            put ["wtf?"] >> pop) []
("wtf?",[])
```

Not bad!

UNIVERSIDAD SIMÓN BOLÍVAR

Y como diseñamos la pila *genérica*...

```
> let f = (sequence $ replicate 2 (push 'o')) >>
          push 'f'
> let q = push 'x' >>
          (sequence $ replicate 2 (push 'u'))
          >> push 'q'
> evalState ((push (runState f [])) >>
            (push (runState q [])) >> get)
            []
[((),"quux"),((),"foo")]
```

Una pila de pilas...

Hay más de una forma de manipular el estado

- `Control.Monad.State.Lazy` provee una versión perezosa – `Control.Monad.State.Strict` provee una versión estricta.
- Enfasis en manipular el estado

```
get    :: MonadState s m => m s
put    :: MonadState s m => s -> m ()
modify :: MonadState s m => (s -> s) -> m ()
gets   :: MonadState s m => (s -> a) -> m a
```

- Ejecutar las transformaciones y acceder al resultado

```
runState  :: State s a -> s -> (a,s)
evalState :: State s a -> s -> a
execState :: State s a -> s -> s
```

UNIVERSIDAD SIMÓN BOLÍVAR

Monad `IO`

Gracias Wadler por los favores concedidos

- `type IO = World -> (a,World)` – No es Haskell válido :-)
- Es un Monad `State` – para alguna definición "razonable" de `World`.
- Programa – cómputo monádico en este Monad `State` especializado

```
main :: IO ()
```

- Por definición de >>=, el mundo se mueve en una sola "dirección" – no hay copias ni viaje en el tiempo.

UNIVERSIDAD SIMÓN BOLÍVAR

Aprovechando el Monad `IO`

- No hay forma de escapar del Monad `IO` – obviamente opaco.
- Pero los cómputos en el Monad `IO` son objetos de primera clase
  - *Construirlos* es diferente a *ejecutarlos*.
  - Pueden combinarse con el resto de los tipos de datos

```
> let msgs = [ putStr "Hello", putStr " World!\n" ]
> :type msgs
msgs :: [IO ()]
> sequence_ msgs
Hello World!
```

- Toda interacción con el mundo se encapsula en el Monad `IO`
  - En el ambiente de ejecución de GHC.
  - En alguna librería externa "ligada" a Haskell (FFI).

UNIVERSIDAD SIMÓN BOLÍVAR

Más combinadores sobre Monads

Listas de Monads

```
sequence :: (Monad m) => [m a] -> m [a]
sequence = foldr mcons (return [])
  where
    mcons p q = p >>=
                \x -> q >>=
                \y -> return (x : y)

> sequence [Just 1, Just 2, Just 3]
Just [1,2,3]
> sequence [print 1,print 2,print 3]
1
2
3
[(),(),()]
```

Lista de Monad se convierte en Monad con lista de resultados.

Más combinadores sobre Monads

Listas de Monads

```
sequence_ :: (Monad m) => [m a] -> m ()
sequence_ = foldr (>>) (return ())

> sequence_ [Just 1, Just 2, Just 3]
Just ()
> sequence_ [print 1,print 2,print 3]
1
2
3
```

Lista de Monad se convierte en Monad conteniendo “void”,
pero los efectos de borde ocurren.

Más combinadores sobre Monads

Listas de Monads – `filter` monádico

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []       = return []
filterM p (x : xs) =
  do
    b  <- p x
    ys <- filterM p xs
    return (if b then (x : ys) else ys)

> filterM (\x -> Just (x>0)) [2,1,0,-1]
Just [2,1]
> filterM (\x -> [x>0]) [2,1,0,-1]
[[2,1]]
```

Lista se convierte en Monad con lista de valores que cumplan un predicado monádico.

Más combinadores sobre Monads

Listas de Monads – `map` monádico

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)

> mapM Just [0,1,2]
Just [0,1,2]
> mapM print [0,1,2]
0
1
2
[(),(),()]
```

Lista de valores se convierte en Monad conteniendo
la lista de resultados de una función monádica.

UNIVERSIDAD SIMÓN BOLÍVAR

Más combinadores sobre Monads

Listas de Monads – `map` monádico

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)

> mapM_ Just [0,1,2]
Just ()
> mapM_ print [0,1,2]
0
1
2
```

Lista de valores se convierte en Monad conteniendo "void"
pero los efectos de borde de la función monádica ocurren
por *cada* elemento de la lista original.

Monad `IO` y combinadores

- `echo` – repetir los argumentos de línea de comandos

Monad IO y combinadores

- echo – repetir los argumentos de línea de comandos

```
main = (intersperse " ") `liftM` getArgs >>=
       mapM_ putStr >> putChar '\n'
```

(IO es Functor – cambien `liftM` por <$> y verán)

- sort – ordenar la entrada

UNIVERSIDAD SIMÓN BOLÍVAR

Monad IO y combinadores

- echo – repetir los argumentos de línea de comandos

```
main = (intersperse " ") `liftM` getArgs >>=
       mapM_ putStr >> putChar '\n'
```

(IO es Functor – cambien `liftM` por <$> y verán)

- sort – ordenar la entrada

```
main = (mapM_ putStrLn . sort . lines) =<<
       getContents
```

UNIVERSIDAD SIMÓN BOLÍVAR

Quiero saber más...

- Documentación de `Control.Monad`
- Documentación de `Control.Monad.State`
- Documentación de `Control.Monad.Writer`
- Documentación de `System.IO`
- Documentación de `System.Random`

UNIVERSIDAD SIMÓN BOLÍVAR