Introducción a Mónadas

Pablo E. "Fidel" Martínez López
fidel@unq.edu.ar

“Había un muro. No parecía importante. (…)

Al igual que todos los muros, era ambiguo, bifacético. Lo que había dentro, o fuera de él, dependía del lado en que uno se encontraba.

Visto desde uno de los lados el muro encerraba un campo baldío de sesenta acres llamado el Puerto de Anarres. (…) Encerraba al universo, dejando fuera a Anarres, libre.”

Los desposeídos

Úrsula K. Le Guin

Mónadas – Overview

- Presentación inspirada en el enfoque usado por Phillip Wadler [1]
- Con mejoras usando la forma de presentar abstracción basada en la técnica de los "recuadros" (parámetros, esquemas de programas)
- Completa la presentación algunos detalles sobre mónadas

Mónadas – Ejemplo básico

- Evaluador básico

```
data E = Cte Float | Div E E

eval :: E -> Float
eval (Cte n)       = ... n …
eval (Div e1 e2)  = ... eval e1 ... eval e2 ...
```

Mónadas – Ejemplo básico

- Evaluador básico

```
data E = Cte Float | Div E E

eval :: E -> Float
eval (Cte n)        = n
eval (Div e1 e2)  = eval e1 / eval e2
```

Mónadas – Ejemplo básico

- Evaluador básico

```
data E = Cte Float | Div E E

eval :: E -> Float
eval (Cte n)       = n
eval (Div e1 e2)  = eval e1 / eval e2
```

(alternativa de codificación)

```
eval :: E -> Float
eval (Cte n)      = n
eval (Div e1 e2) = let v1 = eval e1
                    in let v2 = eval e2
                       in v1 / v2
```

Mónadas – Ejemplo básico

- Evaluador básico

```
data E = Cte Float | Div E E

eval :: E -> Float
eval (Cte n)       = n
eval (Div e1 e2) = let v1 = eval e1
                   in let v2 = eval e2
                      in v1 / v2
```

  - ¿La función es total o parcial?

Mónadas – Modificación 1

- ¿Cómo hacerla total?

```
eval :: E -> Maybe Float
eval (Cte n)       = … n ...
eval (Div e1 e2) =
          … (eval e1) …
                     … (eval e2) ...
```

Mónadas – Modificación 1

- ¿Cómo hacerla total? (2)

```
eval :: E -> Maybe Float
eval (Cte n)        = Just n
eval (Div e1 e2) =
          case (eval e1) of
             Nothing -> Nothing
             Just v1  -> case (eval e2) of
                           Nothing -> Nothing
                           Just v2  -> if v2 == 0
                                        then Nothing
                                        else Just (v1 / v2)
```

Mónadas – Ejemplo básico

- Evaluador básico

```
data E = Cte Float | Div E E

eval :: E -> Float
eval (Cte n)       = n
eval (Div e1 e2) = let v1 = eval e1
                    in let v2 = eval e2
                       in v1 / v2
```

  - ¿Y si quiero llevar una traza de las cuentas?

Mónadas – Modificación 2

- ¿Cómo armar una traza de las cuentas?

```
type Output a = (a, Screen)
type Screen = String

eval :: E -> Output Float
eval (Cte n)   = ... n ...
eval (Div e1 e2) =
     ... (eval e1) ...
             ... (eval e2) ...
```

Mónadas – Modificación 2

- ¿Cómo armar una traza de las cuentas? (2)

```
type Output a = (a, Screen)
type Screen = String

eval :: E -> Output Float
eval (Cte n)   = (n, "")
eval (Div e1 e2) =
    let (v1, o1) = (eval e1)
        in let (v2, o2) = (eval e2)
            in let o3 = printf (formatDiv v1 v2 (v1 / v2))
                in (v1 / v2, o1++o2++o3)
```

Mónadas – Modificación 2

- ¿Cómo armar una traza de las cuentas? (3)
  - (definiciones auxiliares)

```
printf :: String -> Screen
printf msg = msg

formatDiv v1 v2 r = show v1  ++ “/”
                 ++ show v2  ++ “=”
                 ++ show r   ++ “\n”
                        -- show es como toString...
```

Mónadas – Ejemplo básico

- Evaluador básico

```
data E = Cte Float | Div E E

eval :: E -> Float
eval (Cte n)       = n
eval (Div e1 e2) = let v1 = eval e1
                    in let v2 = eval e2
                       in v1 / v2
```

  - ¿Y si quiero contar la cantidad de divisiones?

Mónadas – Modificación 3

- ¿Cómo contar la cantidad de divisiones?

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String

eval' :: E -> StateT Float
eval' (Cte n)  = ... n ...
eval' (Div e1 e2) =
     ... (eval' e1) ...
          ... (eval' e2) ...
```

Mónadas – Modificación 3

- ¿Cómo contar la cantidad de divisiones? (2)

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String

eval' :: E -> StateT Float
eval' (Cte n)  = \s -> (n, s)
eval' (Div e1 e2) =
    \s -> let (v1, s1) = (eval' e1) s
              in let (v2, s2) = (eval' e2) s1
                   in let s3 = inc "div" s2
                        in (v1 / v2, s3)
```

Mónadas – Modificación 3

- ¿Cómo contar la cantidad de divisiones? (3)
  - (definiciones auxiliares)

```
inc ::Variable -> Mem -> Mem
inc “div” (“div”, d) = (“div”, d+1)

eval :: E -> (Float, Mem)
eval e = (eval' e) (“div”, 0)
```

Mónadas

- Alteraciones pequeñas
- Cambios grandes
- ¿Cómo conseguir que los cambios no impacten tanto en el código?
- IDEA: usar la técnica de los “recuadros”
- ¡¡ABSTRAER las diferencias!!

Mónadas – Modificación 1

- Reescribimos y dibujamos los recuadros

```
eval (Cte n) = Just n
eval (Div e1 e2) =
  case (eval e1) of
        Nothing  -> Nothing
        Just v1'  ->
      (\v1 -> case (eval e2) of
                  Nothing -> Nothing
                  Just v2' -> (\v2 -> if v2 == 0
                                       then Nothing
                                       else Just (v1 / v2)
                              ) v2'
      ) v1'
```

Mónadas – Modificación 1

- Reescribimos y dibujamos los recuadros

```
eval (Cte n) = Just n
eval (Div e1 e2) =
  case (eval e1) of
         Nothing  -> Nothing
         Just v1'  ->
            (\v1 -> case (eval e2) of
                          Nothing -> Nothing
                          Just v2' -> (\v2 -> if v2 == 0
                                                then  Nothing
                                                else  Just (v1 / v2)
                                      ) v2'
            ) v1'
```

Mónadas – Modificación 1

- Separamos algunos recuadros...

```
eval (Div e1 e2) =
    (eval e1)
case   m    of
    Nothing   ->  Nothing
    Just v1'  ->  (\v1 -> case (eval e2) of
       k                Nothing  ->  Nothing
                        Just v2' ->  (\v2 -> if v2 == 0
                                              then Nothing
                                              else Just (v1 / v2)
                                     ) v2'
                  )
         v1'
```

Mónadas – Modificación 1

- ...y los ponemos como parámetros

```
eval (Div e1 e2) =
(\m k -> case m of
     Nothing    ->  Nothing
     Just v1'   ->  k v1')
        (eval e1)
        (\v1 -> case (eval e2) of
                  Nothing  ->  Nothing
                  Just v2' ->  (\v2 -> if v2 == 0
                                        then Nothing
                                        else Just (v1 / v2)
                               ) v2'
        )
```

Mónadas – Modificación 1

- ...y los ponemos como parámetros

```
eval (Div e1 e2) =
  (\m k -> case m of
      Nothing     ->  Nothing
      Just v1'    ->  k v1')
        (eval e1)
        (\v1 -> (\m k -> case m of
                    Nothing  ->  Nothing
                    Just v2' ->  k v2')
                      (eval e2)
                      (\v2 -> if v2 == 0
                                then Nothing
                                else Just (v1 / v2))
        )
```

Mónadas – Modificación 1

- Damos nombre a los recuadros

```
returnM :: ??
returnM x   = Just x
bindM :: ??
bindM m k = case m of  Nothing ->   Nothing
                       Just v   ->   k v
raiseError  = Nothing

eval (Cte n)        = returnM n
eval (Div e1 e2) =
    bindM (eval e1)
        (\v1 -> bindM (eval e2)
                   (\v2 -> if v2 == 0
                           then raiseError
                           else returnM (v1 / v2)))
```

Mónadas – Modificación 1

- Reescribimos la sintaxis por comodidad

```
returnM :: ??
returnM x   = Just x
bindM :: ??
bindM m k  = case m of  Nothing -> Nothing
                        Just v   -> k v
raiseError  = Nothing

eval (Cte n)      = returnM n
eval (Div e1 e2) =
      eval e1   `bindM` \v1 ->
      eval e2   `bindM` \v2 ->
      if v2 == 0
       then raiseError
       else returnM (v1 / v2)
```

Mónadas – Modificación 1

- Reescribimos la sintaxis por comodidad

```
returnM :: a -> Maybe a
returnM x    = Just x
bindM :: Maybe a -> (a -> Maybe b) -> Maybe b
bindM m k  = case m of  Nothing -> Nothing
                        Just v    -> k v
raiseError   = Nothing

eval (Cte n)        = returnM n
eval (Div e1 e2) =
         eval e1   `bindM` \v1 ->
         eval e2   `bindM` \v2 ->
         if v2 == 0
          then raiseError
          else returnM (v1 / v2)
```

Mónadas – Modificación 2

- Reescribimos el código y dibujamos los recuadros

```
eval (Cte n) = (\x s -> (x, s)) n
eval (Div e1 e2) =
   \s -> let (v1', s1') = (eval e1) s
             in (\v1 ->
                  \s1 -> let (v2', s2') = (eval e2) s1
                            in (\v2 ->
                                 \s2 ->  let (vd, s3') = inc “div” s2
                                            in (\_ ->
                                                   (\x s3 -> (x, s3)) (v1 / v2)
                                                )  vd s3'
                               )  v2' s2'
                )  v1' s1'
```

Mónadas – Modificación 2

- Rearmamos los recuadros

```
eval (Div e1 e2) =
   (\m k -> \s -> let (v1', s1') = m s
                   in k v1' s1')
      (eval e1)
      (\v1 -> (\m k -> \s1 -> let (v2', ks2')= m s1
                                in k v2' s2')
                 (eval e2)
                 (\v2 -> (\m k -> \s2 -> let (vd, s3') = m s2
                                           in k vd s3')
                            inc "div"
                            (\_  ->  (\x s3 -> (x, s3))  (v1 / v2))))
```

Mónadas – Modificación 2

- Damos nombre a los recuadros

```
returnS :: ??
returnS x = \s -> (x, s)
bindS :: ??
bindS m k = \s -> let (v, s') = m s in k v s'
inc “div” (“div”, v) = ((), (“div”, v + 1))

eval (Cte n)       = returnS n
eval (Div e1 e2) =
    bindS  (eval e1)
           (\v1 -> bindS (eval e2)
                         (\v2 -> bindS inc “div”
                                       (\_ ->  returnS (v1 / v2))))
```

Mónadas – Modificación 2

- Reescribimos la sintaxis por comodidad

```
returnS :: ??
returnS x = \s -> (x, s)
bindS :: ??
bindS m k = \s -> let (v, s') = m s in k v s'
inc “div” (“div”, v) = ((), (“div”, v + 1))

eval (Cte n)       = returnS n
eval (Div e1 e2) =
    eval e1        `bindS` \v1 ->
    eval e2        `bindS` \v2 ->
    inc “div”      `bindS` \_ ->
    returnS (v1 / v2)
```

Mónadas – Modificación 2

- Reescribimos la sintaxis por comodidad

```
returnS :: a -> StateT a
returnS x = \s -> (x, s)
bindS :: StateT a -> (a -> StateT b) -> StateT b
bindS m k = \s -> let (v, s') = m s in k v s'
inc “div” (“div”, v) = ((), (“div”, v + 1))

eval (Cte n)        = returnS n
eval (Div e1 e2) =
    eval e1          `bindS` \v1 ->
    eval e2          `bindS` \v2 ->
    inc “div”        `bindS` \_ ->
    returnS (v1 / v2)
```

Mónadas – Ejemplo básico

- Reescribimos el código y dibujamos los recuadros

```
eval (Cte n) = id n
eval (Div e1 e2) =  let v1' = (eval e1)
                    in (\v1 ->  let v2' = (eval e2)
                                in (\v2 -> id (v1 / v2)
                                   ) v2'
                       ) v1'
```

Mónadas – Ejemplo básico

- Rearmamos los recuadros

```
eval (Cte n) = id n
eval (Div e1 e2) = (\m k -> let v1' = m
                            in k v1')
                   (eval e1)
                   (\v1 -> (\m k -> let v2' = m
                                    in k v2')
                           (eval e2)
                           (\v2 -> id (v1 / v2)))
```

Mónadas – Ejemplo básico

- Damos nombre a los recuadros

```
returnId :: ??
returnId x  = x
bindId :: ??
bindId m k = let v = m in k v

eval (Cte n)      = returnId n
eval (Div e1 e2) = bindId (eval e1)
                          (\v1 -> bindId (eval e2)
                                  (\v2 -> returnId (v1 / v2)))
```

Mónadas – Ejemplo básico

- Damos nombre a los recuadros

```
returnId :: ??
returnId x  = x
bindId :: ??
bindId m k = let v = m in k v

eval (Cte n)       = returnId n
eval (Div e1 e2) = eval e1 `bindId` \v1 ->
                   eval e2 `bindId` \v2 ->
                   returnId (v1 / v2)
```

Mónadas – Ejemplo básico

- Damos nombre a los recuadros

```
returnId :: a -> Id a
returnId x  = x
bindId :: Id a -> (a -> Id b) -> Id b
bindId m k = let v = m in k v

eval (Cte n)         = returnId n
eval (Div e1 e2) = eval e1 `bindId` \v1 ->
                   eval e2 `bindId` \v2 ->
                   returnId (v1 / v2)
```

Mónadas – Definición

- Una mónada es un tipo paramétrico

```
M a
```

con operaciones

```
return :: a -> M a
(>>=) :: M a -> (a -> M b) -> M b
```

que satisfacen las siguientes leyes

```
return x >>= k                 =  k x
m >>= \x -> return x           =  m
m >>= \x -> (n >>= \y -> p)    =  (m >>= \x -> n) >>= \y -> p
```

siempre que `x` no aparezca en `p`

Mónadas – Intuición

- Una mónada incorpora *efectos* a un valor
  - El tipo `M a` incorpora la *información* necesaria
  - `return x` representa a `x` con el *efecto nulo*
  - `(>>=)` *acumula* efectos con *dependencia* de datos
- Es una forma de abstraer comportamientos específicos en un cómputo
  - Observar las diferencias en el código final de cada ejemplo (págs.26,31,36)
  - Idea similar al pattern Strategy en OOP

Mónadas – Intuición

- Cada mónada se diferencia de las demás por sus operaciones adicionales
  - Maybe tiene raiseError
  - StateT tiene incDiv
  - Output tiene printf
  - etc.

Mónadas – y clases

- Haskell define una clase para las mónadas

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

- Para definir Maybe como una mónada se escribe

```
instance Monad Maybe where
    return x = Just x
    m >>= k = case m of
                   Nothing -> Nothing
                   Just x  -> k x
```

Mónadas – y clases

- Se puede usar la clase para pedir una mónada "paramétrica", que proveerá el sistema de tipos

```
eval :: Monad m => E -> m Float
eval (Cte n)       = return n
eval (Div e1 e2) = eval e1 >>= \v1 ->
                   eval e2 >>= \v2 ->
                   return (v1/v2)
```

Do notation

- La do-notation es una forma de abreviar el uso de mónadas para las clases monádicas

```
eval e1            >>= \v1 ->                do v1 <- eval e1
eval e2            >>= \v2 ->                   v2 <- eval e2
imprimir “traza” >>= \_ ->        vs.         imprimir “traza”
return (v1/v2)                                   return (v1/v2)
```

- ¡Observar que es SÓLO *syntactic sugar*!

Mónada IO

- Es una mónada predefinida en Haskell, que captura las operaciones de entrada/salida
- Es un tipo llamado (IO a), con operaciones monádicas, más operaciones primitivas diversas

```
getChar :: IO Char              -- Lee un caracter de teclado
putChar :: Char -> IO ()        -- Escribe un caracter en la pantalla
readFile :: FilePath -> IO String
   -- Lee el contenido de un archivo del disco, en forma de string
writeFile :: FilePath -> String -> IO ()
   -- Graba un archivo con ese nombre, con el contenido dado
```

Mónada IO

- Puede usarse en combinación con do-notation o no, y todos los otros elementos de Haskell

```
fileToUpper :: FilePath -> IO ()
-- Pasa a mayusculas todo el contenido del archivo
fileToUpper fn = do putStrLn “Procesando...”
                    contents <- readFile fn
                    writeFile fn (map toUpper contents)
```

Mónadas – funciones generales

- Pueden definirse muchas funciones de uso general usando sólo la intefase de mónadas

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f mx = do  x <- mx
                 return (f x)

liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f mx my = do x <- mx
                    y <- my
                    return (f x y)

-- Ver el módulo Monad por más funciones...
```

Mónadas – funciones generales

- Estas funciones pueden usarse para definir funciones específicas para la aplicación

```
(</>) :: Monad m => m Float -> m Float -> m Float
(</>) = liftM2 (/)

eval :: Monad m => E -> m Float
eval (Cte n)       = return n
eval (Div e1 e2) = eval e1 </> eval e2
```

Mónadas – funciones generales

- Otras funciones útiles

```
sequence :: Monad m => [ m a ] -> m [a]
sequence []           = return []
sequence (mx:mxs)  = do  x <- mx
                         xs <- sequence mxs
                         return (x:xs)

ej = sequence [ Just 1, Just 2, Just 3, Just 4 ]
```

Mónadas – funciones generales

- Otras funciones útiles

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

ej2 = sequence_ [ putChar 'H', putChar 'o'
                , putChar 'l', putChar 'a' ]
```

Mónadas – funciones generales

- Otras funciones útiles

```
putStr, putStrLn :: String -> IO ()
putStr     msg = sequence_ (map putChar msg)
putStrLn msg = putStr (msg++"\n")

ejFinal = putStrLn "Hello, world!"
```

Mónadas – Conclusiones

- Las mónadas proveen un nivel de abstracción nuevo e iluminador
- La computación secuencial imperativa es solo una de las estrategias posibles de cómputo
- Hay todo un mundo de riquezas monádicas para explorar
- Pensar en abstracto cumple las promesas

Mónadas – Bibliografía

[1] Philip Wadler, Monads for Functional Programming. In [3], pages 24-52.

[2] Mark P. Jones, Functional programming with overloading and higher-order polymorphism. In [3], pages 97-136.

[3] Johan Jeuring and Erik Meijer, editors, *Proceedings of the First International Spring School on Advanced Functional Programming Techniques*. LNCS 925, Springer, 1995.