Introducción a Mónadas

Pablo E. “Fidel” Martínez López

fidel@unq.edu.ar

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 – Modificación 1

- ¿Cómo hacerla total?

```
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 – Modificación 2

- ¿Cómo contar la cantidad de divisiones?

```
type StateT a = State -> (a, State)
type State = Int

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 = incrementar s2
               in (v1 / v2, s3)
```

Mónadas – Modificación 2

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

```
incrementar:: State -> State
incrementar d = d+1

eval' :: E -> (Float, State)
eval' e = (eval e) 0
```

Mónadas – Modificación 3

- ¿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) =
        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 3

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

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

formatDiv v1 v2 r = show v1  ++ "/"
                  ++ show v2  ++ "="
                  ++ show r    ++ "\n"
```

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!!

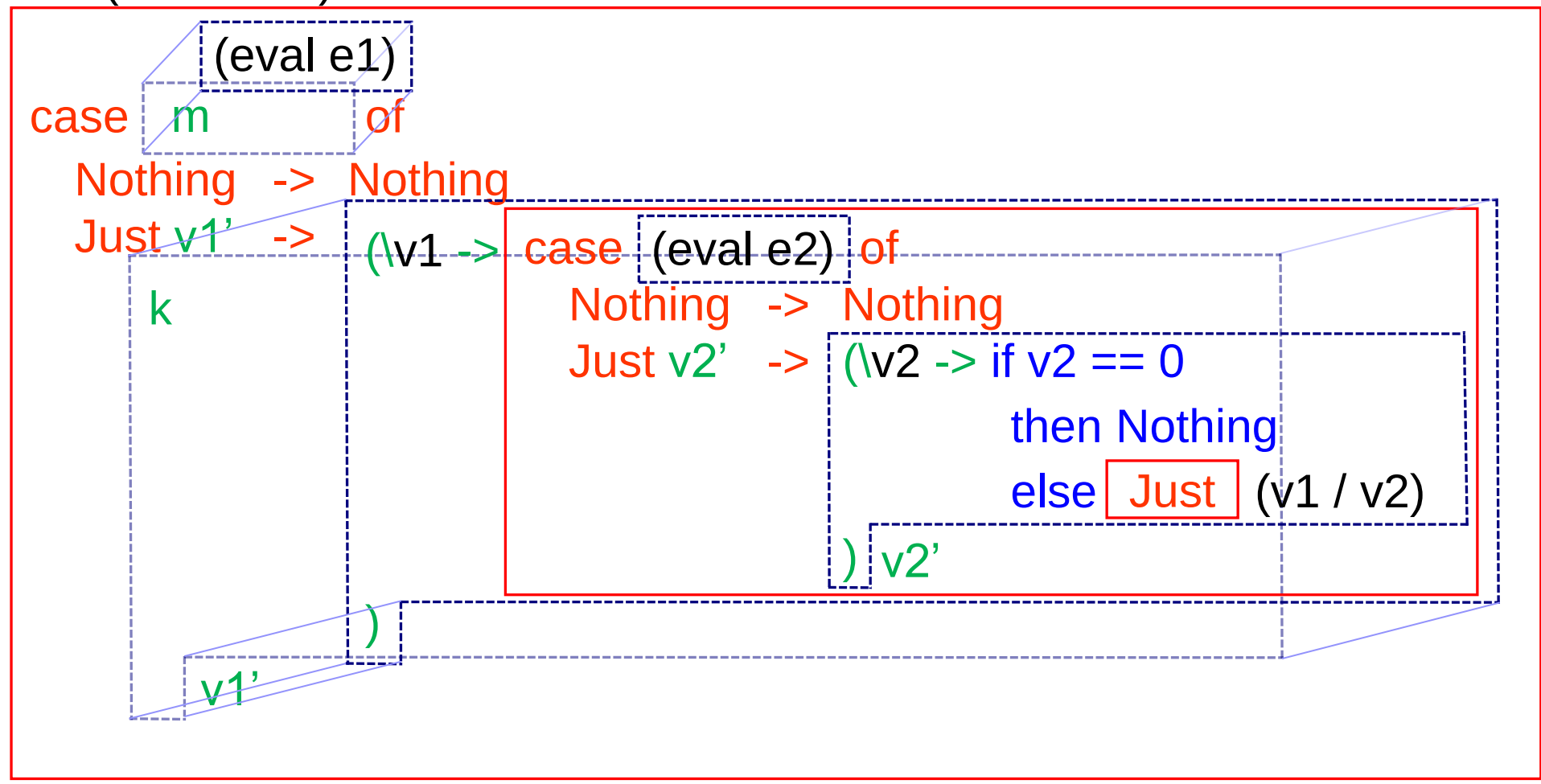
Mónadas – Modificación 1

- Reescribimos el código 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...



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

```
bindM m k = case  m of  Nothing  -> Nothing
                        Just v   -> k v
returnM n  = Just n
failM      = Nothing


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

Mónadas – Modificación 1

- Reescribimos la sintaxis por comodidad

```
bindM m k  = case  m of  Nothing ->  Nothing
                         Just v  ->  k v
returnM n  = Just n
failM      = Nothing

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

Mónadas – Modificación 2

- Reescribimos el código y dibujamos los recuadros

```
eval (Cte n) = (\n' s -> (n', 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') = incrementar s2
                            in (\_ ->
                                (\n' s3 -> (n', 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', s2') = m s1
                              in k v2' s2')
          (eval e2)
          (\v2 -> (\m k -> \s2 -> let (vd, s3') = m s2
                                  in k vd s3')
              incrementar
              (\_  ->  (\n s3 -> (n, s3))  (v1 / v2))))
```

Mónadas – Modificación 2

- Damos nombre a los recuadros

```
bindS m k = \s -> let (v, s') = m s in k v s'
returnS n = \s -> (n, s)
incrementar s = ((), s + 1)

eval (Cte n) = returnS n
eval (Div e1 e2) =
   bindS  (eval e1)
          (\v1 -> bindS (eval e2)
                        (\v2 -> bindS  incrementar
                                       (\_ ->  returnS (v1 / v2))))
```

Mónadas – Modificación 2

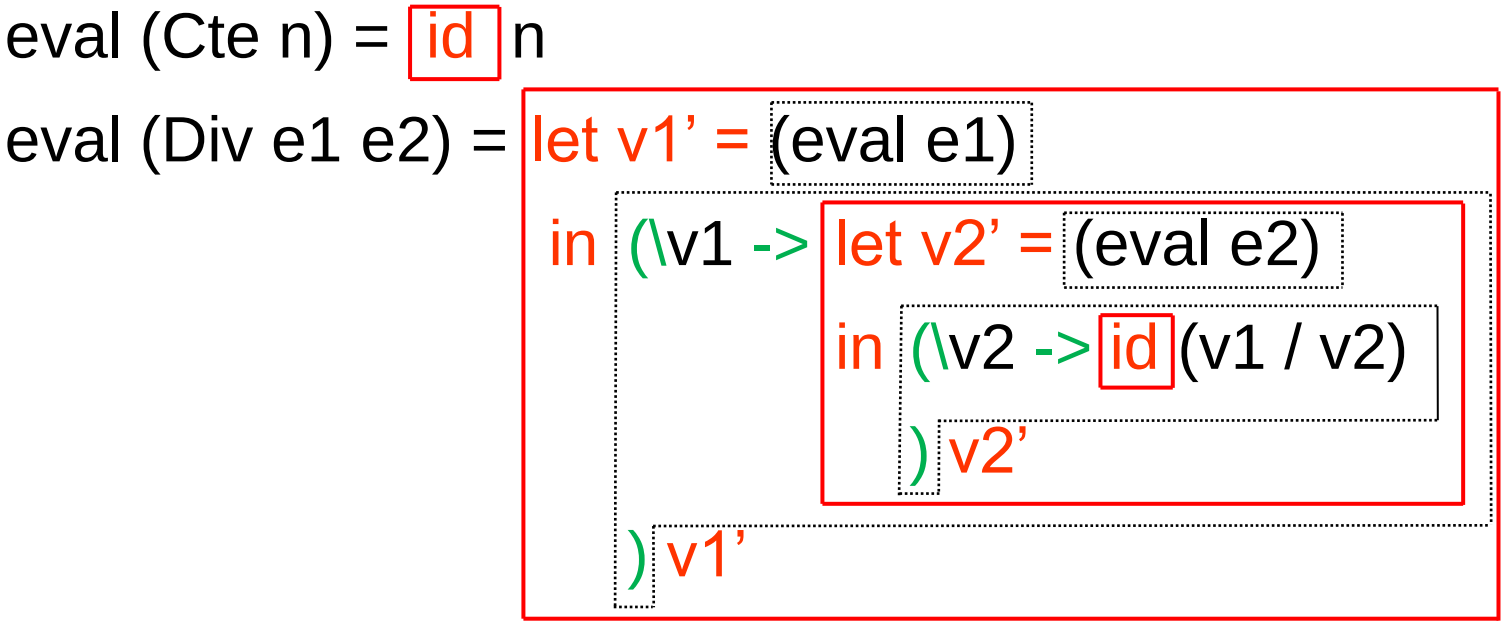
- Reescribimos la sintaxis por comodidad

```
bindS m k     = \s -> let (v, s') = m s in k v s'
returnS n     = \s -> (n, s)
incrementar = \s -> ((), s + 1)

eval (Cte n) = returnS n
eval (Div e1 e2) =
   eval e1                    `bindS` \v1 ->
   eval e2                    `bindS` \v2 ->
   incrementar                `bindS` \_ ->
   returnS (v1 / v2)
```

Mónadas – Ejemplo básico

- Reescribimos el código y dibujamos los recuadros



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

```
bindId m k = let v = m in k v
returnId n  = n

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

```
bindId m k = let v = m in k v
returnId n  = n

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*
  - `(>>=)` *secuencia* 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.12,16,20)
  - Idea similar al pattern Strategy en OOP

Mónadas – Intuición

- Cada mónada se diferencia de las demás por sus operaciones adicionales
  - Maybe tiene fail
  - State tiene incrementar
  - Output tiene imprimir
  - 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 ]
ej2 = sequence [ putChar ‘H’, putChar ‘o’, putchar ‘l’, putChar ‘a’ ]
```

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