Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

ECI 2024

Modelado de efectos computacionales por mónadas

Expresiones con Fallas

```
data Expr = Val Int | Add Expr Expr | Div Expr Expr
```

La operación de división debe controlar el caso excepcional de *división por cero*.

```
data Maybe a = Just a | Nothing

divM       :: Int → Int → Maybe Int
a `divM` b = if b ≡ 0 then Nothing
                      else Just (a `div` b)
```

Evaluador con Fallas

```
eval                :: Expr → Maybe Int
eval (Val n)        = Just n
eval (Add x y) = case eval x of
                      Nothing → Nothing
                      Just a  → case eval y of
                                     Nothing → Nothing
                                     Just b  → Just (a + b)
eval (Div x y) = case eval x of
                      Nothing → Nothing
                      Just a  → case eval y of
                                     Nothing → Nothing
                                     Just b  → a `divM` b
```

Evaluador con Fallas - Applicative

```
eval                :: Expr → Maybe Int
eval (Val n)        = pure n
eval (Add x y)      = (+) <$> eval x <*> eval y
eval (Div x y)      = case eval x of
                        Nothing → Nothing
                        Just a  → case eval y of
                                    Nothing → Nothing
                                    Just b  → a `divM` b
```

Evaluador con Fallas - Applicative

```
eval                :: Expr → Maybe Int
eval (Val n)        = pure n
eval (Add x y)      = (+) <$> eval x <*> eval y
eval (Div x y)      = case eval x of
                        Nothing → Nothing
                        Just a  → case eval y of
                                    Nothing → Nothing
                                    Just b  → a `divM` b
```

No puedo representar la división con Functores Aplicativos, necesito el resultado de una computación para determinar el siguiente efecto.

Capturemos patrones

Definamos:

$$
\begin{aligned}
&\mathit{return} :: a \to \mathit{Maybe}\ a \\
&\mathit{return}\ a = \mathit{Just}\ a
\end{aligned}
$$

$$
\begin{aligned}
&(\gg\!=) :: \mathit{Maybe}\ a \to (a \to \mathit{Maybe}\ b) \to \mathit{Maybe}\ b \\
&m \gg\!= f = \texttt{case}\ m\ \texttt{of} \\
&\qquad\qquad \mathit{Nothing} \to \mathit{Nothing} \\
&\qquad\qquad \mathit{Just}\ a \to f\ a
\end{aligned}
$$

Evaluador con Fallas

Entonces,

$$
\begin{aligned}
&\mathit{eval} && :: \mathit{Expr} \to \mathit{Maybe\ Int} \\
&\mathit{eval}\ (\mathit{Val}\ n) && = \mathit{return}\ n \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) && = \mathit{eval}\ x \gg\!= (\lambda a \to \mathit{eval}\ y \gg\!= (\lambda b \to \mathit{return}\ (a + b))) \\
&\mathit{eval}\ (\mathit{Div}\ x\ y) && = \mathit{eval}\ x \gg\!= (\lambda a \to \mathit{eval}\ y \gg\!= (\lambda b \to a\ `\mathit{divM}`\ b))
\end{aligned}
$$

Evaluador con Fallas

$$
\begin{aligned}
&\mathit{eval} && :: \mathit{Expr} \to \mathit{Maybe\ Int} \\
&\mathit{eval}\ (\mathit{Val}\ n) && = \mathit{return}\ n \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) && = \mathit{eval}\ x \gg\!= \lambda a \to \\
&&& \quad \mathit{eval}\ y \gg\!= \lambda b \to \\
&&& \quad \mathit{return}\ (a + b) \\
&\mathit{eval}\ (\mathit{Div}\ x\ y) && = \mathit{eval}\ x \gg\!= \lambda a \to \\
&&& \quad \mathit{eval}\ y \gg\!= \lambda b \to \\
&&& \quad a\ `\mathit{divM}`\ b
\end{aligned}
$$

La clase Monad

```
class Applicative m ⇒ Monad m where
  (>>=)  :: m a → (a → m b) → m b
  (>>)   :: m a → m b → m b
  return :: a → m a
```

$$m \gg k = m \gg\!= \lambda\_ \rightarrow k$$

La clase Monad

$$\texttt{class}\ \mathit{Applicative}\ m \Rightarrow \mathit{Monad}\ m\ \texttt{where}$$
$$\begin{aligned}(\gg\!=) &:: m\ a \to (a \to m\ b) \to m\ b\\ (\gg) &:: m\ a \to m\ b \to m\ b\\ \mathit{return} &:: a \to m\ a\end{aligned}$$

$$m \gg k = m \gg\!= \lambda\_ \to k$$

Toda mónada es un functor aplicativo que cumple:

- $\mathit{pure} = \mathit{return}$
- $m1\ \texttt{<*>}\ m2 = m1 \gg\!= (\lambda f \to m2 \gg\!= (\lambda x \to \mathit{return}\ (f\ x)))$

La clase Monad

$$\texttt{class}\ \mathit{Applicative}\ m \Rightarrow \mathit{Monad}\ m\ \texttt{where}$$
$$\begin{aligned}(\gg\!=) &:: m\ a \to (a \to m\ b) \to m\ b\\ (\gg) &:: m\ a \to m\ b \to m\ b\\ \mathit{return} &:: a \to m\ a\end{aligned}$$

$$m \gg k = m \gg\!= \lambda\_ \to k$$

Toda mónada es un functor aplicativo que cumple:

- $\mathit{pure} = \mathit{return}$
- $m1\ \texttt{<*>}\ m2 = m1 \gg\!= (\lambda f \to m2 \gg\!= (\lambda x \to \mathit{return}\ (f\ x)))$

No todo functor aplicativo es una mónada

Mónada Maybe

```
data Maybe a = Just a | Nothing

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

Mónada Maybe

```
data Maybe a = Just a | Nothing

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

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just a) = Just (f a)

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

Mónada Maybe

```
data Maybe a = Just a | Nothing

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

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just a) = Just (f a)

instance Applicative Maybe where
  pure  = return
  m1 <*> m2 = m1 >>= (λf -> m2 >>= (λx -> return (f x)))
```

Leyes de mónadas

$$
\begin{aligned}
\mathit{return}\ x \gg\!= f &= f\ x \\
m \gg\!= \mathit{return} &= m \\
(m \gg\!= f) \gg\!= g &= m \gg\!= \lambda x \to (f\ x \gg\!= g)
\end{aligned}
$$

Composición de funciones monádicas

Composición de Kleisli.

$$(\gg\!\!>) :: \mathit{Monad}\ m \Rightarrow (a \to m\ b) \to (b \to m\ c) \to a \to m\ c$$
$$f\ \gg\!\!>\ g = \lambda a \to f\ a \gg\!= g$$

Propiedades:

$$\begin{aligned} \mathit{return} \gg\!\!>\ f &= f \\ f \gg\!\!>\ \mathit{return} &= f \\ f \gg\!\!>\ (g \gg\!\!>\ h) &= (f \gg\!\!>\ g) \gg\!\!>\ h \end{aligned}$$

Se prueban fácilmente usando las leyes de mónadas.

Notación do

$$\texttt{do}\ m = m$$

$$\texttt{do}\ \{x \leftarrow m; m'\} = m \gg\!= \lambda x \rightarrow \texttt{do}\ m'$$

$$\texttt{do}\ \{m; m'\} = m \gg \texttt{do}\ m'$$

Evaluador con Fallas (notación do)

```
eval :: Expr → Maybe Int
eval (Val n)     = return n
eval (Add x y) = do a ← eval x
                    b ← eval y
                    return (a + b)
eval (Div x y)  = do a ← eval x
                     b ← eval y
                     a `divM` b
```

Mónada Either

```
data Either a b = Left a | Right b

instance Monad (Either e) where
  return          = Right
  Left  e >>= _   = Left e
  Right a >>= f   = f a
```

Mónada Either

```
data Either a b = Left a | Right b


instance Monad (Either e) where
  return          = Right
  Left  e >>= _   = Left e
  Right a >>= f   = f a
```

Se corresponde con:

```
instance Applicative (Either e) where
  pure = Right
  Right f <*> Right a = Right (f a)
  Right f <*> Left e  = Left e
  Left e  <*> _       = Left e
```

Mónada Either

```
data Either a b = Left a | Right b

instance Monad (Either e) where
  return          = Right
  Left  e >>= _   = Left e
  Right a >>= f   = f a
```

Pero no con:

```
instance Monoid e => Applicative (Either e) where
  pure = Right
  Right f <*> Right a = Right (f a)
  Left e  <*> Right _ = Left e
  Right _ <*> Left e  = Left e
  Left e  <*> Left e' = Left (e `mappend` e')
```

Mónada Either

```
data Either a b = Left a | Right b

instance Monad (Either e) where
  return          = Right
  Left  e >>= _   = Left e
  Right a >>= f   = f a
```

Pero no con:

```
instance Monoid e => Applicative (Either e) where
  pure = Right
  Right f <*> Right a = Right (f a)
  Left e  <*> Right _ = Left e
  Right _ <*> Left e  = Left e
  Left e  <*> Left e' = Left (e `mappend` e')

Left a <*> Left b = Left (a `mappend` b)
Left a >>= (λf → Left b >>= (λx → return (f x))) = Left a
```

Applicative no monádico

La instancia anterior de *Applicative* para *Either* **no** es una mónada.

```
instance Monoid e ⇒ Monad (Either e) where
  return = Right
  Left e >>= f = ??
  ...
```

Applicative no monádico

La instancia anterior de *Applicative* para *Either* **no** es una mónada.

```
instance Monoid e ⇒ Monad (Either e) where
  return = Right
  Left e >>= f = ??
  ...
```

- No podemos aplicar f en este caso porque sólo se aplica cuando la primera computación retorna un valor (*Right a*).
- Esto no ocurre en la instancia de *Applicative*.

Diferencia entre functores aplicativos y mónadas

La diferencia entre mónadas y functores aplicativos se puede apreciar en los siguientes operadores condicionales:

```
ifM :: Monad m ⇒ m Bool → m a → m a → m a
ifM mb mt me = do b ← mb
                  if b then mt else me
```

No todas computaciones se ejecutan (se elije entre *mt* y *me*)

Diferencia entre functores aplicativos y mónadas

La diferencia entre mónadas y functores aplicativos se puede apreciar en los siguientes operadores condicionales:

```
ifM :: Monad m ⇒ m Bool → m a → m a → m a
ifM mb mt me = do b ← mb
                  if b then mt else me
```

No todas computaciones se ejecutan (se elije entre *mt* y *me*)

```
ifA :: Applicative f ⇒ f Bool → f a → f a → f a
ifA fb ft fe = cond <$> fb <*> ft <*> fe
  where
    cond b t e = if b then t else e
```

Las tres computaciones (*fb*, *ft* y *fe*) se ejecutan y finalmente se elije uno de los resultados.

Mónada de estado

$$\texttt{newtype}\ \mathit{State}\ s\ a = \mathit{State}\ (s \to (a, s))$$

$$\begin{aligned}&\mathit{runState} :: \mathit{State}\ s\ a \to (s \to (a, s))\\&\mathit{runState}\ (\mathit{State}\ f) = f\end{aligned}$$

$$\begin{aligned}&\texttt{instance}\ \mathit{Monad}\ (\mathit{State}\ s)\ \texttt{where}\\&\quad \mathit{return}\ a = \mathit{State}\ \$\ \lambda s \to (a, s)\\&\quad m \gg\!= f = \mathit{State}\ \$\ \lambda s \to \texttt{let}\ (a, s') = \mathit{runState}\ m\ s\\&\qquad\qquad\qquad\qquad\qquad\quad \texttt{in}\ \ \mathit{runState}\ (f\ a)\ s'\end{aligned}$$

Forma alternativa de escribir la definición de $(\gg\!=)$:

$$\begin{aligned}(\mathit{State}\ g) \gg\!= f = \mathit{State}\ \$\ \lambda s \to \texttt{let}\ (a, s') &= g\ s\\ \mathit{State}\ k &= f\ a\\ \texttt{in}\ \ k\ s'&\end{aligned}$$

Funciones sobre estado

```
get :: State s s
get = State $ λs → (s, s)

put :: s → State s ()
put s = State $ λ_ → ((), s)

modify :: (s → s) → State s ()
modify f = get >>= λs → put (f s)

evalState :: State s a → s → a
evalState m s = fst $ runState m s

execState :: State s a → s → s
execState m s = snd $ runState m s
```

Ejemplo: contar número de sumas en una expresión

```
tick :: State Int ()
tick = modify (+1)

evalS :: Expr → State Int Int
evalS (Val n) = return n
evalS (Add e e') = do a ← evalS e
                      b ← evalS e'
                      tick
                      return (a + b)

nroSumas e = execState (evalS e) 0
```

Evaluador con Variables

```
data Expr = Val Int
          | Add Expr Expr
          | Var ID    -- variables
          | Assign ID Expr    -- asignación

eval :: Expr → State (Map ID Int) Int
eval (Val n)      = return n
eval (Add e e')   = do a ← eval e
                       b ← eval e'
                       return (a + b)
eval (Var v)      = do s ← get
                       return (fromJust $ lookup v s)
eval (Assign v e) = do a ← eval e
                       s ← get
                       put (insert v a s)
                       return a
```

Mónada de Estado

```
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
```

```
modify :: MonadState s m ⇒ (s → s) → m ()
modify f = do s ← get
              put (f s)
```

Mónada de Estado

```
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
```

```
modify :: MonadState s m ⇒ (s → s) → m ()
modify f = do s ← get
              put (f s)
```

```
instance MonadState s (State s) where
  get   = State $ λs → (s, s)
  put s = State $ λ_ → ((), s)
```

Mónada List

```
instance Monad [] where
  return x = [x]
  xs >>= f = [y | x <- xs, y <- f x]
                  -- concat (map f xs)
```

Ejemplo: Suma de todos los pares de valores de dos listas

```
sumnd :: Num a => [a] -> [a] -> [a]
sumnd xs ys = do x <- xs
                 y <- ys
                 return (x + y)
```

```
> sumnd [1,3] [4,7]
[5,8,7,10]
```

Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

ECI 2024

Parsers monádicos

Mónada de Parsing

```
newtype Parser a = P (String → [(a, String)])

runP :: Parser a → String → [(a, String)]
runP (P p) = p

instance Monad Parser where
  return a      = P $ λcs → [(a, cs)]
  (P p) >>= f = P $ λcs →
                  concat [runP (f a) cs' | (a, cs') ← p cs]
```

Parsing: combinadores básicos

```
pFail :: Parser a
pFail = P $ λcs → []

item :: Parser Char
item = P $ λcs → case cs of
                   ""     → []
                   (c : cs) → [(c, cs)]

pSat :: (Char → Bool) → Parser Char
pSat p = do c ← item
            if p c then return c else pFail

pSym :: Char → Parser Char
pSym c = pSat (== c)
```

Parsing: Alternativa

$$(<|>) :: \mathit{Parser}\ a \to \mathit{Parser}\ a \to \mathit{Parser}\ a$$
$$(P\ p)\ <|>\ (P\ q) = P\ \$\ \lambda cs \to p\ cs +\!\!+ q\ cs$$

Parsing: Alternativa

$$(<|>) :: Parser\ a \to Parser\ a \to Parser\ a$$
$$(P\ p)\ <|>\ (P\ q) = P\ \$\ \lambda cs \to p\ cs +\!\!+ q\ cs$$

Otra forma de definir el operador de alternativa:

$$(P\ p)\ <|>\ (P\ q) = P\ \$\ \lambda cs \to \texttt{case}\ p\ cs +\!\!+ q\ cs\ \texttt{of}$$
$$\begin{array}{ll} [\,] & \to [\,] \\ (x : xs) & \to [x] \end{array}$$

many y some

p^* (many) cero o más veces p

```
pList :: Parser a → Parser [a]
pList p = do a ← p
             as ← pList p
             return (a : as)
          <|>
          return []
```

p^+ (some) una o más veces p

```
pListP :: Parser a → Parser [a]
pListP p = do a ← p
              as ← pList p
              return (a : as)
```

Ejemplo: digits

```
digit :: Parser Int
digit = do c ← pSat isDigit
           return (ord c − ord '0')

isDigit c = (c ⩾ '0') ∧ (c ⩽ '9')

digits :: Parser [Int]
digits = pListP digit

sumDigits :: Parser Int
sumDigits = do ds ← digits
               return (sum ds)
```

Ejemplo: `number`

```
number :: Parser Int
number = do d ← digit
            number' d

number' :: Int → Parser Int
number' n = do d ← digit
               number' (n * 10 + d)
            <|>
            return n
```

Ejemplo: `number`

```
number :: Parser Int
number = do d ← digit
            number' d

number' :: Int → Parser Int
number' n = do d ← digit
               number' (n * 10 + d)
            <|>
            return n
```

Esto equivale a la siguente definición:

```
number = do (d : ds) ← digits
            return (foldl (⊕) d ds)
         where
           n ⊕ d = n * 10 + d
```

Parser para expresiones

Queremos parsear una expresión y retornar el correspondiente árbol de sintaxis abstracta (AST) de tipo:

$$\texttt{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

Parser para expresiones

Queremos parsear una expresión y retornar el correspondiente árbol de sintaxis abstracta (AST) de tipo:

```
data Expr = Val Int | Add Expr Expr
```

Que tal este parser?

```
expr :: Parser Expr
expr = do e1 ← expr
          pSym '+'
          e2 ← expr
          return (Add e1 e2)
       <|>
       do n ← number
          return (Val n)
```

Parser para expresiones

Queremos parsear una expresión y retornar el correspondiente árbol de sintaxis abstracta (AST) de tipo:

```
data Expr = Val Int | Add Expr Expr
```

Que tal este parser?

```
expr :: Parser Expr
expr = do e1 ← expr
          pSym '+'
          e2 ← expr
          return (Add e1 e2)
       <|>
       do n ← number
          return (Val n)
```

Diverge! La recursividad a la izquierda hace que entre en loop

Parser para expresiones

Para eliminar la recursión a la izquierda debemos basarnos en la siguiente gramática:

$$e ::= n + e \mid n$$

Parser para expresiones

Para eliminar la recursión a la izquierda debemos basarnos en la siguiente gramática:

$$e ::= n + e \mid n$$

El parser queda entonces de la siguiente forma:

```
expr :: Parser Expr
expr = do n ← number
          pSym '+'
          e ← expr
          return (Add (Val n) e)
       <|>
       do n ← number
          return (Val n)
```

Parsing y evaluación de expresiones

$$
\begin{aligned}
evalExpr = \texttt{do}\ & e \leftarrow expr \\
& return\ (eval\ e)
\end{aligned}
$$

Parsing y evaluación de expresiones

```
evalExpr = do e ← expr
              return (eval e)
```

Es posible fusionar las definiciones de *eval* y *expr* y obtener una definición de *evalExp* que computa directamente el valor de la expresión parseada sin generar el AST intermedio:

```
evalExpr :: Parser Int
evalExpr = do n ← number
              pSym '+'
              m ← evalExpr
              return (n + m)
           <|>
           number
```

Parser de un nano XML

```
data XML = Tag Char [XML]


xml :: Parser XML
xml = do    -- se parsea el tag de apertura
          pSym '<'
          name ← item
          pSym '>'
            -- se parsea la lista de XMLs internos
          xmls ← pList xml
            -- se parsea el tag de cierre
          pSym '<'
          pSym '/'
          pSym name    -- se utiliza nombre del tag de apertura
          pSym '>'
          return (Tag name xmls)
```