Programación Funcional Avanzada

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

Modelado de efectos computacionales por mónadas

Evaluador de Expresiones

$$\text{data } Exp = Num\ Int \mid Add\ Exp\ Exp$$

$$\begin{aligned} &eval && :: Exp \to Int \\ &eval\ (Num\ n) && = n \\ &eval\ (Add\ x\ y) && = eval\ x + eval\ y \end{aligned}$$

Supongamos que deseamos agregar un operador de división a nuestras expresiones:

$$\text{data } Exp = Num\ Int \mid Add\ Exp\ Exp \mid Div\ Exp\ Exp$$

Evaluador con Fallas

$$\text{data } \mathit{Exp} = \mathit{Num}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Div}\ \mathit{Exp}\ \mathit{Exp}$$

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

$$\begin{aligned}
&\text{data } \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}\\
&\mathit{divM} \quad :: \mathit{Int} \to \mathit{Int} \to \mathit{Maybe}\ \mathit{Int}\\
&a\ \text{`}\mathit{divM}\text{`}\ b = \text{if } b \equiv 0 \text{ then } \mathit{Nothing}\\
&\qquad\qquad\qquad\qquad\qquad \text{else } \mathit{Just}\ (a\ \text{`}\mathit{div}\text{`}\ b)
\end{aligned}$$

Evaluador con Fallas (2)

```
eval                :: Exp → Maybe Int
eval (Num 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 (3)

Definamos:

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

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

Evaluador con Fallas (4)

Entonces,

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

La clase Monad

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

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

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
```

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.

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

Propiedades:

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

Se prueban fácilmente usando las leyes de mónadas.

Notación do

$$\text{do } m = m$$

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

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

Evaluador con Fallas (notación do)

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Exp} \to \mathit{Maybe\ Int} \\
&\mathit{eval}\ (\mathit{Num}\ n) \quad = \mathit{return}\ n \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathrm{do}\ a \leftarrow \mathit{eval}\ x \\
&\qquad\qquad\qquad\qquad\quad\ \ b \leftarrow \mathit{eval}\ y \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ (a + b) \\
&\mathit{eval}\ (\mathit{Div}\ x\ y) \ = \mathrm{do}\ a \leftarrow \mathit{eval}\ x \\
&\qquad\qquad\qquad\qquad\quad\ \ b \leftarrow \mathit{eval}\ y \\
&\qquad\qquad\qquad\qquad\quad\ \ a\ `\mathit{divM}`\ b
\end{aligned}
$$

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

throw :: e -> Either e a
throw = Left

catch :: Either e a -> (e -> Either e a) -> Either e a
catch (Left  e) h = h e
catch (Right a) _ = Right a
```

Excepciones - Ejemplo

$$\text{data } Err = DivZero \mid ...$$

$$\begin{aligned} &divE :: Int \to Int \to Either\ Err\ Int \\ &a\ `divE`\ b \mid b == 0 \quad = throw\ DivZero \\ &\qquad\qquad \mid otherwise = return\ (a\ `div`\ b) \end{aligned}$$

Supongamos que el evaluador retorna 0 para recuperar una falla por división por cero.

$$\begin{aligned} &eval :: Exp \to Either\ Err\ Int \\ &eval\ e = ... \end{aligned}$$

$$\begin{aligned} &evalExc\ e = catch\ (eval\ e)\ handler \\ &\quad \text{where } handler\ DivZero = return\ 0 \end{aligned}$$

Funciones sobre Mónadas (ver Control.Monad)

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

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

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

Funciones sobre Mónadas (ver Control.Monad)

$$
\begin{aligned}
&addM :: (Num\ a, Monad\ m) \Rightarrow m\ a \to m\ a \to m\ a \\
&addM\ m\ m' = \mathrm{do}\ x \leftarrow m \\
&\qquad\qquad\qquad\quad y \leftarrow m' \\
&\qquad\qquad\qquad\quad return\ (x + y)
\end{aligned}
$$

$$
\begin{aligned}
&liftM2 :: (a \to b \to c) \to m\ a \to m\ b \to m\ c \\
&liftM2\ f\ m\ m' = \mathrm{do}\ x \leftarrow m \\
&\qquad\qquad\qquad\quad y \leftarrow m' \\
&\qquad\qquad\qquad\quad return\ (f\ x\ y)\}
\end{aligned}
$$

$$
addM = liftM2\ (+)
$$

Mónada de estado

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

$$runState :: State\ s\ a \to (s \to (a, s))$$
$$runState\ (State\ f) = f$$

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

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

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

Funciones sobre estado

$$get :: State\ s\ s$$
$$get = State\ \$\ \lambda s \rightarrow (s, s)$$

$$put :: s \rightarrow State\ s\ ()$$
$$put\ s = State\ \$\ \lambda\_ \rightarrow ((), s)$$

$$modify :: (s \rightarrow s) \rightarrow State\ s\ ()$$
$$modify\ f = get \gg\!= \lambda s \rightarrow put\ (f\ s)$$

$$evalState :: State\ s\ a \rightarrow s \rightarrow a$$
$$evalState\ m\ s = fst\ \$\ runState\ m\ s$$

$$execState :: State\ s\ a \rightarrow s \rightarrow s$$
$$execState\ m\ s = snd\ \$\ runState\ m\ s$$

Ejemplo: contar número de sumas en una expresión

$$
\begin{aligned}
&\mathit{tick} :: \mathit{State}\ \mathit{Int}\ () \\
&\mathit{tick} = \mathit{modify}\ (+1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{evalS} :: \mathit{Exp} \to \mathit{State}\ \mathit{Int}\ \mathit{Int} \\
&\mathit{evalS}\ (\mathit{Num}\ n) = \mathit{return}\ n \\
&\mathit{evalS}\ (\mathit{Add}\ e\ e') = \mathsf{do}\ a \leftarrow \mathit{evalS}\ e \\
&\qquad\qquad\qquad\qquad\qquad\quad b \leftarrow \mathit{evalS}\ e' \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{tick} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (a + b)
\end{aligned}
$$

$$
\mathit{nroSumas}\ e = \mathit{execState}\ (\mathit{evalS}\ e)\ 0
$$

Evaluador con Variables

```
data Exp = Num Int
         | Add Exp Exp
         | Var ID              -- variables
         | Assign ID Exp       -- asignación

eval :: Exp → State (Map ID Int) Int
eval (Num 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)
```

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

Parsers monádicos

Mónada de Parsing

$$\text{newtype } Parser\ a = P\ (String \to [(a, String)])$$

$$runP :: Parser\ a \to String \to [(a, String)]$$
$$runP\ (P\ p) = p$$

$$\begin{aligned}
&\text{instance } Monad\ Parser \text{ where} \\
&\quad return\ a \quad\quad = P\ \$\ \lambda cs \to [(a, cs)] \\
&\quad (P\ p) \gg\!= f = P\ \$\ \lambda cs \to \\
&\qquad\qquad\qquad\qquad concat\ [runP\ (f\ a)\ cs' \mid (a, cs') \leftarrow p\ cs]
\end{aligned}$$

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: combinadores básicos

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

Otra forma de definir el operador de alternativa:

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

$$\begin{aligned}&pList :: Parser\ a \to Parser\ [a]\\ &pList\ p = \text{do}\ \{a \leftarrow p; as \leftarrow pList\ p; return\ (a : as)\}\\ &\qquad\quad <|>\\ &\qquad\quad return\ [\,]\end{aligned}$$

Ejemplo: *digits*

$$
\begin{aligned}
&\mathit{digit} :: \mathit{Parser}\ \mathit{Int} \\
&\mathit{digit} = \mathrm{do}\ c \leftarrow \mathit{pSat}\ \mathit{isDigit} \\
&\qquad\qquad \mathit{return}\ (\mathit{ord}\ c - \mathit{ord}\ \text{'0'})
\end{aligned}
$$

$$
\mathit{isDigit}\ c = (c \geqslant \text{'0'}) \wedge (c \leqslant \text{'9'})
$$

$$
\begin{aligned}
&\mathit{digits} :: \mathit{Parser}\ [\mathit{Int}] \\
&\mathit{digits} = \mathit{pList}\ \mathit{digit}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sumDigits} :: \mathit{Parser}\ \mathit{Int} \\
&\mathit{sumDigits} = \mathrm{do}\ \mathit{ds} \leftarrow \mathit{digits} \\
&\qquad\qquad\qquad \mathit{return}\ (\mathit{sum}\ \mathit{ds})
\end{aligned}
$$

Ejemplo: *number*

$$
\begin{aligned}
&\mathit{number} :: \mathit{Parser}\ \mathit{Int}\\
&\mathit{number} = \mathit{number}'\ 0
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{number}' :: \mathit{Int} \to \mathit{Parser}\ \mathit{Int}\\
&\mathit{number}'\ n = \mathsf{do}\ d \leftarrow \mathit{digit}\\
&\qquad\qquad\quad \mathit{number}'\ (n * 10 + d)\\
&\qquad\qquad <|>\\
&\qquad\qquad \mathit{return}\ n
\end{aligned}
$$

Esto equivale a la siguente definición:

$$
\begin{aligned}
&\mathit{number} = \mathsf{do}\ \mathit{ds} \leftarrow \mathit{digits}\\
&\qquad\qquad\quad \mathit{return}\ (\mathit{foldl}\ (\oplus)\ 0\ \mathit{ds})\\
&\qquad\quad \mathsf{where}\\
&\qquad\qquad n \oplus d = n * 10 + d
\end{aligned}
$$

Parser para expresiones

Dada una expresión queremos retornar el correspondiente árbol de sintaxis abstracta de tipo:

$$\text{data } Exp = Num\ Int \mid Add\ Exp\ Exp$$

Que tal este parser?

```
expresion :: Parser Exp
expresion = do e1 <- expresion
               pSym '+'
               e2 <- expresion
               return (Add e1 e2)
            <|>
            do n <- number
               return (Num n)
```

Problema! 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:

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

El parser queda entonces de la siguiente forma:

```
expresion :: Parser Exp
expresion = do n ← number
               pSym '+'
               e ← expression
               return (Add (Num n) e)
            <|>
            do n ← number
               return (Num n)
```

Parsing y evaluador de expresiones

```
evalExp = do e ← expresion
             return (eval e)
```

Es posible fusionar las definiciones de *eval* y *expresion* y obtener una definición de *evalExp* que computa directamente el valor de la expresión reconocida por el parser sin generar el árbol intermedio:

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