Programación Declarativa 2020-II

Una introducción a las Mónadas

Favio Ezequiel Miranda Perea Javier Enríquez Mendoza

Departamento de Matemáticas
Facultad de Ciencias UNAM

6 de mayo de 2020

Lenguajes funcionales

puros vs. impuros

- Los lenguajes funcionales pueden ser puros o impuros.
- Los lenguajes puros son implementaciones directas de sistemas de cálculo lambda.
- Los lenguajes impuros son implementaciones de extensiones del cálculo lambda con efectos laterales como asignación, excepciones o continuaciones.
- Puros: Haskell
- Impuros: Scheme, ML

(Des)ventajas de los lenguajes puros

- Razonamiento matemático disponible (transparencia referencial)
- Se benefician de la evaluación perezosa
- Ineficientes y menos expresivos, a veces.
- Proporcionan máxima flexibilidad al estar disponibles y accesibles todos el conjunto de datos en un proceso particular.
- La esencia de un algoritmo puede perderse en el proceso de transporte de datos desde la creación hasta el punto de uso.

Estudio de un caso

Evaluador de un micro lenguaje de programación

- Expresiones aritméticas simples:

  `e ::= n | e + e`

- Sintaxis abstracta:

  `t ::= Num n | Suma e e`

- Intérprete: función de evaluación `eval :: Exp -> Nat`

```
eval (Num n) = n

eval (Sum e1 e2) = (eval e1) + (eval e2)
```

- Ejemplo:

```
eval (Suma (Num 2) (Suma (Num 7) (Num3))) = 12
```

Estudio de un caso

Evaluador de un micro lenguaje de programación

- Expresiones aritméticas simples con jiribilla:

  `e ::= n | e / e`

- Sintaxis abstracta:

  `t ::= Num n | Div e e`

- Intérprete: función de evaluación `eval :: Exp -> Int`

```
eval (Num n) = n

eval (Div e1 e2) = (eval e1) / (eval e2)
```

- Ejemplo:

```
eval ( Div (Div (Num 1972) (Num 2)) (Num 23)) = 42
eval ( Div (Num 1972) (Num 0))  =  mamá!!!!
```

Manejo de errores con Maybe

Evaluador de un micro lenguaje de programación

- Expresiones aritméticas simples con doble jiribilla:

  `e ::= n | e / e`
- Sintaxis abstracta:

  `t ::= Num n | Div e e`
- Intérprete: función de evaluación `eval :: Exp -> Maybe Int`

```
eval (Num 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)
```

Otras extensiones posibles

- Manejo de errores: es necesario modificar cada llamada recursiva para verificar y manejar errores (ya se hizo con Maybe, se sigue la misma idea para excepciones más informativas).
- Conteo de operaciones: es necesario modificar cada llamada recursiva para pasar un contador explícito.
- Trazas de ejecución: es necesario modificar cada llamada recursiva para ir construyendo las trazas.
- En un lenguaje impuro el código original se mantiene agregando:
  - El uso de excepciones
  - Un contador global
  - Un efecto de sálida
- En Haskell podemos usar una mónada.

Evaluador

Conteo del número de divisiones

- Tipo transformador de estados:

```
data M a = State -> (a, State)

type State = Int
```

- Intérprete: `eval :: Exp -> M Int`

```
eval (Num n) s = (n,s)

eval (Div e1 e2) s = let (v1,s1) = eval e1 s
                         (v2,s2) = eval e2 s1
                     in
                      (v1/v2, s2 + 1)
```

Evaluador

Salida con las trazas de evaluación

- Tipo de efectos de salida:

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

- Intérprete: `eval :: Exp -> M Int`

```
eval (Num n) = (line (Num n) n, n)

eval (Div e1 e2) =
  let (l1,v1) = eval e1
      (l2,v2) = eval e2
  in
   (l1 ++ l2 ++
        displayline (Div e1 e2) (v1/v2), (v1/v2))

displayline :: Exp -> Int -> Output
displayline e n = ``eval(`` ++ show e ++ ``) <= ``
            ++ show n ++ ``\n''
```

Programacion monadica

La idea general

- Evaluador original: `eval :: Exp -> Int`
- Evaluador con efectos laterales: `eval :: Exp -> M Int`
- En cada caso el tipo `M Int` es un tipo de cómputos que devuelven un entero causando un efecto lateral.
- La idea general en la programación monádica es reemplazar una función de tipo `a -> b` por una función de tipo `a -> M b`
- El tipo `M b` es lo que se conoce como una mónada y debe cumplir ciertas propiedades específicas que permitan la interacción entre valores simples de tipo `a` y valores monádicos de tipo `M a`.

La clase mónada

Haskell

```
Prelude> :i Monad
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
  -# MINIMAL (>>=) #-
        -- Defined in GHC.Base
instance Monad (Either e) -- Defined in Data.Either
instance Monad [] -- Defined in GHC.Base
instance Monad Maybe -- Defined in GHC.Base
instance Monad IO -- Defined in GHC.Base
instance Monad ((->) r) -- Defined in GHC.Base
instance Monoid a => Monad ((,) a) -- Defined in GHC.Base
```

Evaluador monádico

Mónada identidad

```
type M a = a

return :: a -> M a

return x = x

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

c >>= f   =   f c
```

Evaluador monádico

Mónada identidad

- ```
  eval :: Exp -> M Int
  eval (Num n) = return n

  eval (Div e1 e2) = (eval e1) >>=
                     (\ v1 -> ( eval e2 >>=
                                (\ v2. return (v1/v2))
                              )
  ```

- Idea: evaluar e_1 ligar el resultado a la variable $v1$, evaluar e_2 ligar el resultado a la variable $v2$, devolver la división de $v1$ entre $v2$.
  ```

Evaluador monádico

Mónada de estado

```
type M a = State -> (a,State)

type State = Int

return :: a -> M a

return x  = \ s -> (x,s)

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

c >>= f = \ s ->
              let (v,s') = c s in
               f v s'
```

Evaluador monádico

Mónada de estado

```
eval :: Expr -> M Int

eval (Num n)  = return n

eval (Div e1 e2)  = eval e1 >>=
                        ( \ z1-> eval e2 >>=
                            (\ z2 ->
                                (\ s -> (div z1 z2,s+1))))
```

- Idea: ??

Los operadores monádicos

- Retorno: convierte un valor en un cómputo que lo devuelve y no hace nada más.

  ```
  return :: a -> M a
  ```

- Ligado: aplica una función que devuelve un cómputo al resultado de otro cómputo

  ```
  (>>=) :: m a -> (a -> m b) -> m b
  ```

Los operadores monádicos

- Ligado irrelevante: liga dos cómputos, ignorando el primero.

```
(>>) :: m a -> m b -> m b
c1 >> c2  =  >>= ( \ _ -> c2)
```

- Falla: convierte una cadena en un cómputo erroneo informativo.

```
fail :: String -> M a
fail = error
```

Leyes monádicas

- Primera ley monádica: el único efecto de `return` es pasar su valor:

```
return e >>= f   =   f e
```

- Segunda ley monádica: `return` es elemento identidad derecho de `(>>=)`:

```
p >>= return   =   p
```

- Tercera ley monádica: el operador `(>>=)` es asociativo:

```
(p >>= f) >>= g   =   p >>= (\ x -> (f x >>= g))
```

La notación do

Evaluador monádico

```
eval (Num n) = return n

eval (Div e1 e2) = do r1 <- eval e1
                      r2 <- eval e2
                      return (r1/r2)
```

- La notación do hace que un programa monádico sea más claro y conciso.
- Las instrucciones do son azucar sintáctica.

Notación do

- Las principales instrucciones do se definen como sigue:

```
do {p} = p
do {p;stmts} = p >> do {stmts}
do {x <- p;stmts} = p >>= \ x -> do {stmts}
```

- p denota a una acción (un proceso que devuelve un tipo monádico)
- stmts denota a una secuencia *no vacía* de enunciados que pueden ser acciones o instrucciones de la forma `x <- p` (que no se consideran acciones).
- Luego entonces, expresiones como `do x <-p` son erroneas, así como `do { }`.

Leyes monádicas

notación do

- Primera ley monádica (identidad izquierda): el único efecto de `return` es pasar su valor:

```
(return e >>= f) = f e

do { x <- return e; f x } = do { f e }
```

- Segunda ley monádica (identidad derecha): `return` es elemento identidad derecho de `>>=`

```
(p >>= return) = p

do { x <- p; return x } = p
```

Leyes monádicas

notación do

- Tercera ley monádica (pseudoasociatividad):

```
((p >> = f) >>= g) = p >> = (\ x -> (f x >> = g))

do {y <- do {x <- p; f x} ; g y} =

do {x <- p ; do {y <- f x ; g y}} =
```

- Sobre el anidamiento de do:

```
do { do { stmts }} = do {stmts}
do { stmts1 ; do {stmts2}} = do {stmts1 ; stmts2 }
```

- Entonces la tercera ley monádica puede simplificarse como:

```
do {y <- do {x <- p; f x} ; g y} =
do { x <- p ; y <- f x ; g y }
```