Modelando efectos computacionales con mónadas

11/11/2024

Repaso

En Haskell las mónadas se implementan mediante la clase:

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

donde cada instancia debe satisfacer las siguientes leyes:

```
return a >>= k =  k a

m >>= return  =  m

m >>=(\x -> k x >>= h) = (m >>= k)>>= h
```

Vimos dos ejemplos de mónadas: Maybe y []. La primera nos permitía estructurar programas que podían fallar y la segunda combinar computaciones no deterministas.

Objetivos

- Veremos cómo utilizar mónadas para modelar efectos computacionales, como:
  - manejar un error,
  - llevar un estado y
  - leer los valores asociados a variables en un entorno
- Comprobaremos que las mónadas son una forma efectiva de estructurar código.

Evaluador de expresiones

```
data Exp =   Lit Int          -- enteros
           | Add Exp Exp      -- sumas
           | Div Exp Exp      -- divisiones

eval :: Exp -> Int
eval (Lit n) = n
eval (Add t u) = eval t + eval u
eval (Div t u) = div (eval t) (eval u)
```

Evaluador con manejo de error

```
eval :: Exp -> Maybe Int
eval (Lit n) = Just n
eval (Add t u) = case (eval t) of
     Nothing -> Nothing
     Just x  -> case (eval u) of
                  Nothing -> Nothing
                  Just y  -> Just (x+y)

eval (Div t u) = case (eval t) of
  Nothing -> Nothing
  Just x -> case (eval u) of
              Nothing -> Nothing
              Just y -> if y==0
                        then Nothing
                        else Just (div x y)
```

Usando la mónada Maybe

```
throw :: Maybe a
throw = Nothing

eval :: Exp -> Maybe Int
eval (Lit n)   = return n

eval (Add t u) = eval t >>= \x ->
                 eval u >>= \y ->
                 return (x+y)

eval (Div t u) = eval t >>= \x ->
                 eval u >>= \y ->
                 if y==0
                   then throw
                   else return (div x y)
```

Notación do

En general un programa monádico tiene la siguiente estructura:

```
m1 >>= \x1 ->
m2 >>= \x2 ->
...
return (f x1 x2 ... xn)
```

Haskell provee una sintaxis especial que mejora la legibilidad:

```
do x1 <- m1
   x2 <- m2
   ...
   return (f x1 x2 ... xn)
```

Evaluador monádico con notación do

```
eval :: Exp -> Maybe Int
eval (Lit n)    = return n

eval (Add t u) = do x <- eval t
                    y <- eval u
                    return (x+y)

eval (Div t u) = do x <- eval t
                    y <- eval u
                    if y==0
                     then throw
                     else return (div x y)
```

Evaluador 2: Cuenta las operaciones

```
eval2 :: Exp -> (Int, Int)

eval2 (Lit n)    = (n, 0)

eval2 (Add t u) = let (m, cm) = eval2 t
                      (n, cn) = eval2 u
                  in (n + m, cm + cn + 1)

eval2 (Div t u) = let (m, cm) = eval2 t
                      (n, cn) = eval2 u
                  in (div n m, cm+cn+1)
```

Mónada Acum

- El nuevo evaluador retorna una tupla como resultado, donde los valores de las llamadas recursivas que corresponden a la segunda componente del par se suman.

- ¿Podremos definir una mónada que capture éste efecto?

```
newtype Acum a = Ac {runAc :: (a, Int)}


instance Monad Acum where
  return x = Ac (x, 0)
  Ac(x, n) >>= f = let Ac(x', n') = f x
                   in Ac (x', n + n')
```

Evaluador 2 monádico

```
tick :: Acum ()
tick = Ac ((), 1)

eval2 :: Exp -> Acum Int
eval2 (Lit n)    = return n

eval2 (Add t u) = do x <- eval2 t
                     y <- eval2 u
                     tick
                     return (x+y)

eval2 (Div t u) = do x <- eval2 t
                     y <- eval2 u
                     tick
                     return (div x y)
```

Mónada Writer

- El valor de la segunda componente de la tupla en el tipo Acµm puede no ser un entero. Una mónada más general se obtiene a partir del siguiente tipo de datos.

```
newtype Writer w a =
        Writer {runW :: (a,w)}
```

- Para dar la instancia de `Monad` para `Writer` y probar que es una mónada necesitamos `w` debe ser un monoide:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

donde `mempty` es neutro para `mappend` y `mappend` es asociativa.

Mónada Writer

```
instance (Monoid w) =>
         Monad (Writer w) where
  return x = Writer (x, mempty)
  (Writer (x,v)) >>= f =
        let (Writer (y, v')) = f x
        in Writer (y, mappend v v')
```

Evaluador con variables

Extendemos el lenguaje de expresiones con variables con nombres de tipo `String`.

```
data Exp =   ...
           | Var String

type Env = String -> Int

eval3 :: Exp -> Env -> Int
eval3 (Lit n) e     = n
eval3 (Var v ) e    = e v
eval3 (Plus t u) e = eval3 t e + eval3 u e
eval3 (Div t u) e   = div (eval3 t e)
                          (eval3 u e)
```

Mónada Reader

- Modela computaciones que llevan un entorno.

```
newtype Reader a = {runR :: (Env -> a)}

instance Monad Reader where
  return x = Reader (\_ -> x)
  Reader h >>= f =
        Reader (\e -> runR (f (h e)) e)

ask :: Reader Env -- devolver entorno
ask  = Reader id
```

- Ver instancia de Monad de `((->) e)` en `Control.Monad.Instances`.

Evaluador con variables monádico

```
eval3 :: Exp -> Reader Int
eval3 (Lit n)      = return n

eval3 (Var v )     = do e <- ask
                        return (e v)

eval3 (Plus t u) = do x <- eval3 t
                      y <- eval3 u
                      return (x+y)

eval3 (Div t u)  = do x <- eval3 t
                      y <- eval3 u
                      return (div x y)
```

Combinamos los 3 evaluadores

1. Damos un tipo de datos que capture los efectos de las 3 mónadas.

```
newtype M a =
      M {runM :: Env -> Maybe (a, Int)}
```

2. Ejercicio: Dar la instancia de `M` teniendo en cuenta el propósito de cada mónada.

   Solución:

```
instance Monad M where
  return x = M (\_ -> Just (x, 0))
  M h >>= f =
    M (\e -> case h e of
       Nothing -> Nothing
       Just (a,m)-> case runM (f a) e of
            Nothing -> Nothing
            Just (b, n) -> Just (b, m+n))
```

Operaciones de M

```haskell
-- lanza error
throw :: M a
throw = M (\_ -> Nothing)

-- obtiene entorno
ask :: M Env
ask = M (\e -> Just (e, 0))

-- acumula 1
tick :: M ()
tick = M (\_ -> Just ((), 1))
```

Evaluador monádico

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

eval (Var v )     = do e <- ask
                       return (e v)

eval (Plus t u) = do x <- eval t
                     y <- eval u
                     tick
                     return (x+y)

eval (Div t u) = do x <- eval t
                    y <- eval u
                    if y==0
                     then throw
                     else return (div x y)
```

Observaciones

- Cada evaluador tiene una estructura similar, la cual pudo abstraerse usando la noción de mónada.
- En cada evaluador se introdujo un tipo de cómputo, donde el constructor monádico `M` representó cómputos:
  - que pueden fallar
  - que llevan un acumulador
  - que leen de un entorno
- Las funciones de tipo `a -> b` se reemplazaron por funciones de tipo `a -> M b`, las cuales toman un valor de tipo `a` y devuelve un valor de tipo `b` con un posible efecto adicional capturado por `M`.

Operaciones de las mónadas

Las operaciones soportadas por cada mónada pueden definirse en una clases. Por ejemplo:

```
class Monad m => MonadThrow m where
   throw :: m a

class Monad m => MonadAcum m where
   tick :: m ()

class Monad m => MonadReader m where
   ask :: m Env

eval :: (MonadThrow m ,
         MonadAcum m,
         MonadReader m) => Exp -> m Int
...
```

Observaciones

- Una mónada es una abstracción, para que la abstracción esté bien usada las funciones que usan la mónada sólo deben usar la interfaz de la misma, es decir:
  - `return`, `(>>=)` y
  - operaciones propias de la mónada (`throw`, `tick`, `ask`).

- Las mónadas permiten escribir código más modular y reusable pero también pueden utilizarse otras estructuras, tal vez menos intuitivas que las mónadas como `arrows` o `functores aplicativos`.

Bibliografía

G. Hutton.
Programming in haskell (2nd ed).
2007.

P. Wadler.
Monads for functional programming.
1995.