CI4251 - Programación Funcional Avanzada
Tarea 3

Ernesto Hernández-Novich
86-17791
<emhn@usb.ve>

Junio 5, 2013

Transformadores de Monads

Escribir aplicaciones complejas con Haskell eventualmente conduce a la necesidad de modelar una combinación de ambiente de evaluación, estado, manejo de errores e interacción con el mundo exterior. Todo esto operando de forma modular y que permita refinar la funcionalidad.

Considere nuevamente el evaluador de expresiones de la Tarea 2.

```
data Exp = Const Int
         | Var String
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp
         deriving (Eq,Show)

type Env = DM.Map String Int

env :: Env
env = DM.fromList [ ("foo",42), ("bar",69),
                    ("baz",27), ("qux",0) ]

eval (Const i) = i
eval (Var n)   = fromJust (DM.lookup n env)
eval (Add l r) = (eval l) + (eval r)
eval (Sub l r) = (eval l) - (eval r)
eval (Mul l r) = (eval l) * (eval r)
eval (Div l r) = (eval l) `div` (eval r)

ex1 = (Const 21)
ex2 = (Var "foo")
ex3 = (Mul (Const 2) ex1)
ex4 = (Add (Mul ex2 (Const 5)) (ex3))
ex5 = (Add (Div (Const 42) (Const 2)) (Mul (Const 3) (Const 7)))
```

En esta tarea nos interesa construir una aplicación completa de evaluación que combine todas las funcionalidades que fueron implantadas por separado, agregando eventualmente interacción con el mundo exterior.

La tarea está organizada para que Ud. construya paso a paso una "pila" de transformadores de Monads hasta conseguir esa aplicación simple. La construcción paso a paso pretende, por un lado, que Ud. comprenda la relación que hay entre la evaluación pura y la evaluación monádica, y por otro, desarrollar destrezas en cuanto al orden idóneo de composición. [1]

La tarea está concebida para ser completada en orden, siendo cada paso dependiente del anterior. En cada paso, se construirá un evaluador que agrega funcionalidad al anterior gracias a la transformación de Monads. Seguramente habrá duplicación de código, el cual Ud. puede factorizar si lo desea aunque no es necesario.

Evaluación Pura a Monad `Identity` (3 puntos)

El Monad `Identity` corresponde a la evaluación pura, sólo que estableciendo una secuencia específica de pasos. Una de las mejores prácticas para el desarrollo con transformadores de Monads sugiere definir un tipo de datos *alias* para el transformador, de manera que comenzaremos con

```
type Eval1 a = Identity a
```

Aprovechando esa definición, se desea que Ud. convierta el evaluador puro en uno monádico sobre el monad `Identity`. Para eso, Ud. debe implantar las funciones

```
eval1  :: Env -> Exp -> Eval1 Int
evalM1 :: Eval1 a -> a
```

Note que `eval1` recibe *explícitamente* el ambiente de evaluación y la expresión a evaluar. Entonces, `eval1` *construye* la evaluación monádica, mientras que `evalM1` la *ejecuta*, i.e.

```
ghci> evalM1 (eval1 env ex2)
42
```

Escriba `evalM1` en el estilo *point-free* para su único argumento.

[1] El orden de composición que se presentará en esta tarea no es necesariamente el mejor, pero es probablemente uno de los más razonables al estructurar transformadores de monads en la práctica.

Manejo de Errores (4 puntos)

Una vez lograda la transformación de evaluación pura a monádica, queremos proteger el evaluador para manejar los errores correctamente. Vamos a reportar la misma clase de errores que en la Tarea 2, a saber

```
data ExpError = DivisionPorCero
              | NumeroDeMalaSuerte
              | VariableNoExiste String
              deriving (Show)

instance Error ExpError
```

Para lograrlo, es necesario transformar la operación monádica expresada con `Identity` para incorporarle el manejo de errores con el Monad `Error` manifestado en el tipo de datos `Either`. Una vez más, comenzaremos por definir el tipo de datos *alias* para el Monad combinado

```
type Eval2 a = ErrorT ExpError Identity a
```

Aprovechando esa definición, se desea que Ud. convierta el evaluador monádico simple sobre el Monad `Identity`, en uno que sea capaz de reportar errores de manera segura. Para eso, Ud. debe implantar las funciones

```
eval2  :: Env -> Exp -> Eval2 Int
evalM2 :: Eval2 a -> Either ExpError a
```

Note que `eval2` aún recibe *explícitamente* el ambiente de evaluación y la expresión a evaluar. Entonces, `eval2` *construye* la evaluación monádica combinada, mientras que `evalM2` la *ejecuta*, i.e.

```
ghci> evalM2 (eval2 env ex2)
Right 42
ghci> evalM2 (eval2 env (Const 13))
Left NumeroDeMalaSuerte
```

Escriba `evalM2` en el estilo *point-free* para su único argumento.

Abstracción del Ambiente de Evaluación (4 puntos)

Una vez lograda la transformación que asegura la evaluación monádica con manejo de errores, queremos abstraer el Ambiente de Evaluación para que sea "sólo lectura" desde el principio de la evaluación.

Ahora, es necesario transformar la operación monádica expresada con `Identity` y con manejo de errores, para agregarle la gestión del Ambiente de Ejecución usando el Monad `Reader`, de manera tal que el ambiente particular se defina *una sola vez* al comenzar la evaluación, y no sea necesario pasarlo explícitamente en cada evaluación parcial.

Definiremos el tipo de datos *alias* para el Monad combinado

```
type Eval3 a = ReaderT Env (ErrorT ExpError Identity) a
```

Aprovechando esa definición, se desea que Ud. convierta el evaluador monádico simple con manejo de errores, en uno que abstraiga el ambiente de evaluación desde su inicio. Para eso, Ud. debe implantar las funciones

```
eval3  :: Exp -> Eval3 Int
evalM3 :: Env -> Eval3 a -> Either ExpError a
```

Note que `eval3` ya no recibe el ambiente de evaluación, sino solamente la expresión a evaluar. Entonces, `eval3` *construye* la evaluación monádica combinada, mientras que `evalM3` es el que toma el ambiente inicial y lo establece *implícitamente* para la *ejecución*, i.e.

```
ghci> evalM3 env (eval3 ex2)
Right 42
ghci> evalM3 env (eval3 (Const 13))
Left NumeroDeMalaSuerte
```

Escriba `evalM3` en el estilo *point-free* para el *segundo* argumento.

Incorporando Estado Mutable (4 puntos)

Una vez lograda la transformación para tener una evaluación monádica con manejo de errores y ambiente de referencia constante, queremos incorporar estado mutable con el cual calcular algunas estadísticas sobre el proceso de evaluación, idénticas a las que se calcularon en la Tarea 2.

Ahora, es necesario transformar la operación monádica expresada con `Identity`, con manejo de errores y con ambiente de referencia, para agregarle la gestión del estado mutable usando el Monad `State`, de manera tal que el estado inicial se defina *una sola vez* al comenzar la evaluación, y no sea necesario pasarlo explícitamente en cada evaluación parcial.

Comenzaremos por definir

```
data EvalState = EvalState { adds, subs, muls, divs, vars,
                             tabs :: Int
                           }
               deriving (Show)

initialState = EvalState {
                           adds = 0,
                           subs = 0,
                           muls = 0,
                           divs = 0,
                           vars = 0,
                           tabs = 0
                         }
```

para disponer de un tipo de datos en el cual modelar el estado mutable para el conteo de operaciones[2] y el estado inicial.

Definiremos el tipo de datos *alias* para el Monad combinado

```
type Eval4 a = ReaderT Env
                 (ErrorT ExpError
                    (StateT EvalState Identity)) a
```

Aprovechando esa definición, se desea que Ud. convierta el evaluador monádico simple con manejo de errores y ambiente de evaluación implícito desde su inicio, a uno que incorpore el estado mutable y calcule las estadísticas durante el cómputo. Para eso, Ud. debe implantar las funciones

```
eval4  :: Exp -> Eval4 Int
evalM4 :: Env -> EvalState -> Eval4 a
       -> (Either ExpError a,EvalState)
```

[2] Note el atributo adicional (`tabs`) que usaremos más adelante para indicar anidamiento.

Note que `eval4` ya no recibe el ambiente de evaluación, sino solamente la expresión a evaluar. Entonces, `eval4` *construye* la evaluación monádica combinada, mientras que `evalM4` es el que toma el ambiente y estado iniciales para establecerlos *implícitamente* en la *ejecución*, i.e.

```
ghci> evalM4 env initialState (eval4 ex2)
(Right 42,
EvalState {adds = 0, subs = 0, muls = 0, divs = 0,
vars = 1, tabs = 0})
ghci> evalM4 env initialState (eval4 (Const 13))
(Left NumeroDeMalaSuerte,
EvalState {adds = 0, subs = 0, muls = 0, divs = 0,
vars = 0, tabs = 0})
```

Escriba `evalM4` en el estilo *point-free* para su *tercer* argumento.

Traza de operaciones (4 puntos)

Una vez lograda la transformación para tener una evaluación monádica con manejo de errores, ambiente de referencia constante y estado mutable, queremos incorporar la traza de operaciones en la cual acumular un registro del proceso de evaluación, idéntico al que se recopiló en la Tarea 2.

Ahora, es necesario transformar la operación monádica expresada con `Identity`, con manejo de errores, con ambiente de referencia, y estado mutable, para agregarle la gestión de acumulación usando el Monad `Writer`, de manera tal que comience con una bitácora vacía en la cual acumular la traza, pero no sea necesario pasarla explícitamente en cada evaluación parcial.

Como es costumbre, emplearemos el Monoide `Data.Sequence` para acumular la traza, de modo que definiremos

```
type ExpLog = DS.Seq String
```

Definiremos el tipo de datos *alias* para el Monad combinado

```
type Eval5 a = ReaderT Env
                 (ErrorT ExpError
                    (WriterT ExpLog
                       (StateT EvalState Identity))) a
```

Aprovechando esa definición, se desea que Ud. convierta el evaluador monádico simple con manejo de errores, ambiente de evaluación y estado

mutable implícitos desde su inicio, a uno que incorpore la traza de operaciones y las registre como en la Tarea 2. Para eso, Ud. debe implantar las funciones

```
eval5  :: Exp -> Eval5 Int
evalM5 :: Env -> EvalState -> Eval5 a
       -> ((Either ExpError a,ExpLog),EvalState)
```

Note que `eval5` continúa sin recibir el ambiente de evaluación, solamente la expresión a evaluar. Entonces, `eval5` *construye* la evaluación monádica combinada, mientras que `evalM5` es el que toma el ambiente y estado iniciales para establecerlo *implícitamente* en la *ejecución*, i.e.

```
ghci> evalM5 env initialState (eval5 ex2)
((Right 42,
fromList ["Exp: Var \"foo\" -> Val: 42"]),
EvalState {adds = 0, subs = 0, muls = 0,
divs = 0, vars = 1, tabs = 0})
ghci> evalM5 env initialState (eval5 (Const 13))
((Left NumeroDeMalaSuerte,
fromList ["Exp: Const 13 -> Val: 13"]),
EvalState {adds = 0, subs = 0, muls = 0,
divs = 0, vars = 0, tabs = 0})
```

Escriba `evalM5` en el estilo *point-free* para su *tercer* argumento.

Interacción con el mundo exterior (4 puntos)

Una vez lograda la transformación para tener una evaluación monádica con manejo de errores, ambiente de referencia constante, estado mutable y traza de operaciones, incorporar funcionalidad que le permita interactuar con el mundo exterior, esto es, comportamiento en el Monad `IO`.

Como no existe el "transformador de IO", la manera de lograrlo es convertir la evaluación base pura en impura, para lo cual se *sustituye* el Monad `Identity` por el Monad `IO`. El efecto que queremos lograr es que el evaluador imprima la expresión por pantalla representando el nivel de anidamiento usando espacios en blanco (una suerte de árbol abstracto de la expresión), además de producir todos los resultados combinados. Ud. debe aprovechar el atributo `tabs` en el estado mutable, para calcular la indentación adecuada en cada paso de evaluación.

Definiremos el tipo de datos *alias* para el Monad combinado

```
type Eval6 a = ReaderT Env
                  (ErrorT ExpError
                     (WriterT ExpLog
                        (StateT EvalState IO))) a
```

Aprovechando esa definición, se desea que Ud. convierta el evaluador monádico simple con manejo de errores, ambiente de evaluación, estado mutable implícitos desde su inicio y traza de operaciones, a uno que imprima, *en cada paso de evaluación*, el elemento evaluado particular (número, símbolo u operador) reflejando su nivel de anidamiento en la expresión de manera visual usando espacios en blanco. Para eso, Ud. debe implantar las funciones

```
eval6  :: Exp -> Eval6 Int
evalM6 :: Env -> EvalState -> Eval6 a
          -> IO ((Either ExpError a,ExpLog),EvalState)
```

Note que `evalM6` produce sus resultados en el monad `IO`, así que es de invocación directa en GHCi o en un programa ejecutable autónomo. Entonces, `eval6` es el que *construye* la evaluación monádica combinada, mientras que `evalM6` es el que toma el ambiente y estado iniciales para establecerlo *implícitamente* en la *ejecución*, i.e.

```
ghci> evalM6 env initialState (eval5 ex4)
+
  *
    42
    5
  *
    2
    21
((Right 252,
fromList ["Exp: Var \"foo\" -> Val: 42",
"Exp: Const 5 -> Val: 5",
"Exp: Mul (Var \"foo\") (Const 5) -> Val: 210",
"Exp: Const 2 -> Val: 2",
"Exp: Const 21 -> Val: 21",
"Exp: Mul (Const 2) (Const 21) -> Val: 42",
"Exp: Add (Mul (Var \"foo\") (Const 5))
(Mul (Const 2) (Const 21)) -> Val: 252"]),
EvalState {adds = 1, subs = 0, muls = 2, divs = 0, vars = 1, tabs = 0})
```

Note que las primeras líneas de la salida que exhiben la indentación son efectos de borde (generados con `print` o `putStrLn` según Ud. prefiera), mientras que el resto no es mas que el resultado de la evaluación.

Otra manera de verlo es ejecutar

```
ghci> evalM6 env initialState (eval6 ex4) >> putStrLn "done!"
+
  *
    42
    5
  *
    2
    21
done!
```

que ignora los resultados y sólo muestra los efectos de borde.

Escriba `evalM6` en el estilo *point-free* para su *segundo* argumento.

Para pensar después de todo (3 puntos)

1. El evaluador puro tiene la firma

   ```
   eval1 :: Env -> Exp -> Eval1 Int
   ```

   y se invoca gracias a la función auxiliar `evalM1` de la forma

   ```
   ghci> evalM1 (eval1 env (Const 42))
   42
   ```

   ¿Puede *generalizar* la firma de la función para que **no** sea necesaria la función auxiliar, es decir, qué pueda invocarse

   ```
   ghci> eval1 env (Const 42)
   42
   ```

   directamente? Escriba la firma aquí sin incluirla en el código fuente

   ```
   eval1 :: {- firma generalizada -}
   ```

2. A pesar de haber construido una pila de transformadores de Monad, probablemente habrá notado que no le resultó necesario escribir (y si lo hizo, **quítelas**) aplicaciones de la función `lift`[3] para tener acceso a los valores en los Monad subyacentes. ¿Cuál es la explicación? [4]

3. Cuando agregamos estado mutable, decidimos utilizar el tipo

```
type Eval4 a = ReaderT Env
                 (ErrorT ExpError
                    (StateT EvalState Identity)) a
```

   ¿Qué habría pasado si hubiesemos definido

```
type Eval4 a = ReaderT Env
                 (StateT EvalState
                    (ErrorT ExpError Identity)) a
```

   como tipo para el evaluador?

[3] Seguro tuvo que usar `litfIO`, pero no me refiero a esas aplicaciones.
[4] Use the source, Luke!