Práctica 6 - Mónadas

1. Demostrar que los siguientes tipos son mónadas.

$$\textbf{newtype}\ \mathsf{Id}\ a\ = \mathsf{Id}\ a$$
$$\textbf{data}\ \mathsf{Maybe}\ a = \mathsf{Nothing} \mid \mathsf{Just}\ a$$

Es decir,

a) Dar la instancia de Monad para cada uno de ellos.

b) Demostrar que para cada instancia valen las leyes de las mónadas.

2. Demostrar que el constructor de tipo [ ] (lista) es una mónada.

3. Se desea modelar computaciones con un estado global s. Para esto se define el siguiente tipo de datos e instancia de mónada:

```
newtype State s a = St {runState :: s → (a, s)}
instance Monad (State s) where
  return x     = St (λs → (x, s))
  (St h) >>= f = St (λs → let (x, s') = h s
                          in runState (f x) s')
```

a) Probar que la instancia efectivamente define una mónada.

b) Definir operaciones $\mathsf{set} :: s \to \mathsf{State}\ s\ ()$ y $\mathsf{get} :: \mathsf{State}\ s\ s$ que permiten actualizar el estado y leerlo, respectivamente.

4. Dado el tipo:

$$\textbf{data}\ \mathsf{Tree}\ a = \mathsf{Leaf}\ a \mid \mathsf{Branch}\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)$$

Y su correspondiente instancia Functor:

```
instance Functor Tree where
  fmap f (Leaf a)       = Leaf (f a)
  fmap f (Branch l r) = Branch (fmap f l) (fmap f r)
```

a) La función $\mathsf{numTree} :: \mathsf{Tree}\ a \to \mathsf{Tree\ Int}$, permite numerar las hojas de un árbol de izquierda a derecha. Definir la función auxiliar $\mathsf{mapTreeNro} :: (a \to \mathsf{Int} \to (b, \mathsf{Int})) \to \mathsf{Tree}\ a \to \mathsf{Int} \to (\mathsf{Tree}\ b, \mathsf{Int})$ de forma que la siguiente definición de numTree sea correcta:

```
numTree :: Tree a → Tree Int
numTree t = fst (mapTreeNro update t 0)
  where update a n = (n, n + 1)
```

b) Para generalizar el caso del ítem a, se puede pensar que en lugar Int se lleva un estado de tipo s, quedando una función con la forma:

$$\mathsf{mapTreeSt} :: (a \to s \to (b, s)) \to \mathsf{Tree}\ a \to s \to (\mathsf{Tree}\ b, s)$$

Esto conduce directamente a la utilización de la mónada State s, con la siguiente función:

$$\mathsf{mapTreeM} :: (a \to \mathsf{State}\ s\ b) \to \mathsf{Tree}\ a \to \mathsf{State}\ s\ (\mathsf{Tree}\ b)$$

Definir con notación **do** la función mapTreeM.

5. La clase Monoid clasifica los tipos que son monoides y está definida de la siguiente manera

```
class Monoid m where
  mempty  :: m
  mappend :: m → m → m
```

Se requiere que las instancias hagan cumplir que mappend sea asociativa, y que mempty sea un elemento neutro de mappend por izquierda y por derecha.

a) Probar que String es un monoide.

b) Probar que el siguiente constructor de tipos es una mónada, (asumiendo que el parámetro w es un monoide).

```
newtype Output w a = Out (a, w)
```

c) Dar una instancia diferente de Monad para el mismo tipo. Esto prueba que un mismo tipo de datos puede tener diferentes instancias de mónadas.

d) Definir una operación write :: Monoid $w \Rightarrow w \to$ Output w ().

e) Usando Output String, modificar el evaluador monádico básico de la teoría para agregar una traza de cada operación. Por ejemplo:

```
> eval (Div (Con 14) (Con 2))
El término (Con 14) tiene valor 14
El término (Con 2) tiene valor 2
El término (Div (Con 14) (Con 2)) tiene valor 7
7
```

6. Sea M una mónada. Dados los operadores:

```
(>>)  :: M a → M b → M b
(>>=) :: M a → (a → M b) → M b
```

a) De ser posible, escribir ($\gg$) en función de ($\gg=$).

b) De ser posible, escribir ($\gg=$) en función de ($\gg$).

7. Las siguientes funciones se definen sobre un constructor de tipos m arbitrario:

a) Definir la función sequence :: Monad $m \Rightarrow [m\ a] \to m\ [a]$, de manera tal que sequence xs evalúe todos los valores monádicos en la lista xs, de izquierda a derecha, y devuelva un valor en la misma mónada que calcule la lista de los "resultados" de dichas evaluaciones.

Ejemplo: Si $xs = [m_1, m_2]$, entonces

```
sequence xs = m1 >>= λx1 →
              m2 >>= λx2 →
              return [x1, x2]
```

b) Definir la función liftM :: Monad $m \Rightarrow (a \to b) \to m\ a \to m\ b$, tal que liftM $f\ x$ aplique f al contenido de x dentro de la mónada m.

Ejemplo: liftM sum (Just [1 . . 10]) = Just 55

Notar que liftM coincide con fmap (ejercicio de la práctica anterior).

c) Definir la función liftM2 :: Monad $m \Rightarrow (a \to b \to c) \to m\ a \to m\ b \to m\ c$, tal que liftM2 $f\ m_1\ m_2$ aplique la función binaria f a los contenidos de m_1 y m_2 dentro de la mónada m.

Ejemplo: liftM2 ($\wedge$) [True, False] [True, True] = [True, True, False, False]

d) Expresar sequence como un foldr. (*Sugerencia:* Usar liftM2.)

8. Dado el siguiente tipo de datos:

$$\textbf{data}\ \mathsf{Error}\ er\ a = \mathsf{Raise}\ er \mid \mathsf{Return}\ a$$

a) Demostrar que es una mónada.

b) Dar una definición total de las siguientes funciones, utilizando la mónada Error String:

i. head y tail, correspondientes a las operaciones sobre listas;

ii. push y pop, correspondientes a las operaciones sobre pilas.

9. Se desea implementar un evaluador para un lenguaje sencillo, cuyos términos serán representados por el tipo de datos:

$$\textbf{data}\ \mathsf{T} = \mathsf{Con\ Int} \mid \mathsf{Div\ T\ T}$$

Se busca que el evaluador cuente la cantidad de divisiones, y reporte los errores de división por cero. Se plantea el siguiente tipo de datos para representar una mónada de evaluación:

$$\textbf{newtype}\ \mathsf{M}\ s\ e\ a = \mathsf{M}\ \{\mathsf{runM} :: s \rightarrow \mathsf{Error}\ e\ (a, s)\}$$

y entonces el evaluador puede escribirse de esta manera:

```
eval             :: T → M Int String Int
eval (Con n)     = return n
eval (Div t1 t2) = do v1 ← eval t1
                      v2 ← eval t2
                      if v2 ≡ 0 then raise "Error: Division por cero."
                                else do modify (+1)
                                        return (v1`div`v2)
```

y el cómputo resultante se podría ejecutar mediante una función auxiliar:

```
doEval   :: T → Error String (Int, Int)
doEval t = runM (eval t) 0
```

a) Dar la instancia de la mónada M s e.

b) Determinar el tipo de las funciones raise y modify, y dar su definición.

c) Reescribir eval sin usar notación **do** y luego expandir las definiciones de $\gg=$, return, raise y modify, para obtener un evaluador no monádico.

Ejercicios Adicionales

10. El tipo de datos Cont r a representa *continuaciones* en las que dado el resultado de una función (de tipo a) y la continuación de la computación ($a \to r$), devuelve un valor en r. Probar que Cont r es una mónada. Ayuda: Guiarse por los tipos.

$$\textbf{data}\ \mathsf{Cont}\ r\ a = \mathsf{Cont}\ ((a \to r) \to r)$$

11. Dado el siguiente tipo de datos:

$$\textbf{data}\ \mathsf{M}\ m\ a = \mathsf{Mk}\ (m\ (\mathsf{Maybe}\ a))$$

a) Probar que para toda mónada m, M m es una mónada.

b) Definir una operación auxiliar throw :: Monad $m \Rightarrow$ M m a que lanza una excepción.

c) Dada la mónada de estado StInt y el siguiente tipo N.

$$\textbf{data}\ \mathsf{StInt}\ a = \mathsf{St}\ (\mathsf{Int} \to (a, \mathsf{Int}))$$
$$\textbf{type}\ \mathsf{N}\ a = \mathsf{M}\ \mathsf{StInt}\ a$$

Definir operaciones get :: N Int y put :: Int $\to$ N (), que lean y actualizen (respectivamente) el estado de la mónada.

d) Usando N, definir un intérprete mónadico para un lenguaje de expresiones aritméticas y una sola variable dado por el siguiente AST.

```
data Expr = Var
          | Con Int
          | Let  Expr Expr
          | Add Expr Expr
          | Div  Expr Expr
```

El constructor Var corresponde a dereferenciar la única variable, Con corresponde a una constante entera, Let t, a asignar a la única variable el valor de la expresión t, y Add y Div corresponden a la suma y la división respectivamente. La variable tiene un valor inicial 0. El intérprete debe ser una función total que devuelva el valor de la expresión y el valor de la (única) variable. Por ejemplo, si llamamos a la única variable $\square$, la expresión: **let** $\square = (2+3)$ **in** $\square/7$, queda representada en el AST por la expresión:

```
Let (Add (Con 2) (Con 3)) (Div Var (Con 7)
```