CI4251 - Programación Funcional Avanzada
Tareas 7 y 8

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

Julio 1, 2010

1. Diseño y construcción de `Monad` (6 puntos)

Los cómputos en un Monad de juegos disponen de una cantidad de "vidas". Considere la siguiente clase de tipos

```
class Monad m => GameMonad m where
  extraLife  :: m ()
  getLives   :: m Int
  checkPoint :: m a -> m a
  die        :: m a
```

La función `getLives` retorna la cantidad de vidas restantes, mientras que `extraLife` agrega una vida adicional. La función `die` disminuye la cantidad de vidas en uno.

Por último, `checkPoint` permite continuar el "juego" desde el punto en el que es invocada, siempre y cuando se cuente con vidas disponibles. Esto es, si `m` es algún cómputo monádico (el "juego"), existen tres casos posibles para el cómputo `checkPoint m`:

- `m` se calcula hasta el final, posiblemente aumentado o perdiendo vidas, pero nunca muriendo. El resultado de `checkPoint m` es el mismo de `m` (el "juego" continúa).
- El resultado del cómputo `m` indica que el jugador "muere", pero como le quedan vidas, el cómputo `m` se reintenta desde el principio.
- El resultado del cómputo `m` indica que el jugador "muere", y como no le quedan vidas, el cómputo `checkPoint m` debe fallar.

Note que `m` es *cualquier* cómputo monádico, posiblemente una pila arbitraria de Monads compuestos, cuyo resultado final puede ser exitoso (con resultado que nos interesa) o fallido (con justificación que también nos interesa).

Implante un Monad concreto definiendo el tipo de datos `Game`, con las instancias necesarias de `Monad` y `GameMonad`. No puede utilizar las *funciones* de `Control.Monad.*`.

También es necesario que defina la función

```
runGame :: Game a         -- El juego particular.
        -> Int            -- Cantidad inicial de vidas.
        -> Maybe (a,Int) -- Resultado y vidas restantes.
```

Para orientarse en la implantación, considere que una vez escrita, debe permitir escribir los siguientes fragmentos de programa:

```
printLives :: (GameMonad m, MonadIO m) => String -> m ()
printLives = do
  n <- getLives
  liftIO $ putStrLn $ s ++ " " ++ show n

test1 :: (GameMonad m, MonadIO m) => m ()
test1 = checkPoint $ do
  printLives "Vidas: "
  die
  liftIO $ putStrLn "Ganamos!"

lastChance :: GameMonad m => m ()
lastChance = do
  n <- getLives
  if n == 1 then return ()
            else die

test2 :: (GameMonad m, MonadIO m) => m String
test2 = checkPoint $ do
  printLives "Inicio"
  n <- getLives
  if n == 1
    then do
      liftIO $ putStrLn "Final"
      return "Victoria!"
    else do
      checkPoint $ do
        printLives "Checkpoint anidado"
        lastChance
      extraLife
      printLives "Vida extra!"
      die
```

Los cuales tienen que poder evaluarse como sigue, con los resultados mostrados: [1]

```
ghci> runGameT test1 3
Vidas: 3
Vidas: 2
Vidas: 1
Nothing
ghci> runGameT test2 3
Inicio 3
Checkpoint anidado 3
Checkpoint anidado 2
Checkpoint anidado 1
Vida extra! 2
Inicio 1
Finish
Just ("Victoria!",1)
```

2. Diseño y construcción de transformadores de Monads (4 puntos)

Implante el transformador de Monads `GameT` que agrega a cualquier Monad `m` los comportamientos del Monad `Game`. Además de las instancias para `Monad` y `GameMonad`, es necesario que provea la instancia para `MonadTrans`. No puede utilizar las *funciones* de `Control.Monad.*`.

Para que los ejemplos de la Pregunta 1 puedan ejecutarse, necesitará la instancia

```
instance MonadIO m => MonadIO (GameT m) where
  liftIO = lift . liftIO
```

y una implantación para la función

```
runGameT :: Monad m => GameT m a -> Int -> m (Maybe (a, Int))
```

con un comportamiento idéntico al de la función `runGame` implantada en la Pregunta 1.

[1] La función `runGameT` corresponde a la Pregunta 2, sin embargo lo interesante de estos ejemplos es que se oriente para deducir cuál es la funcionalidad necesaria en su tipo de datos y en el Monad concreto.