CI4251 - Programación Funcional Avanzada
Entrega Tarea 2

Brian-Marcial Mendoza
07-41206
<07-41206@usb.ve>

Mayo 22, 2015

Autómatas Finitos No Determinísticos

```
import Control.Applicative ((<$>), (<*>))
import Control.Monad
import Control.Monad.RWS
import Control.Monad.Error
import qualified Data.List as List (subsequences)
import qualified Data.Sequence as Seq
import qualified Data.Set as DS
import Data.Char
import Data.Either
import Test.QuickCheck
import Test.QuickCheck.Monadic
```

Se importaron además de las librerías especificadas las siguientes tres:

- Control.Applicative: para hacer uso de <*>y <$>.
- Data.List: para hacer uso de subsequences.
- Test.QuickCheck.Monadic: para hacer las propiedades de los NFA usando operaciones monádicas.

El tipo NFANode no fue modificado.

```
newtype NFANode = Node Int
                deriving (Eq,Ord)

instance Show NFANode where
   show (Node i) = "q" ++ show i
```

El tipo de las transiciones no fue modificado.

```
data Transition = Move   { from, to :: NFANode, sym :: Char }
                | Lambda { from, to :: NFANode }
                deriving (Eq,Ord)

instance Show Transition where
  show (Move f t i) = show f ++
                      " -" ++ show i ++ "-> " ++
                      show t
  show (Lambda f t) = show f ++ " ---> " ++ show t
```

El tipo NFA no fue modificado.

```
data NFA = NFA {
                   sigma   :: (DS.Set Char),
                   states  :: (DS.Set NFANode),
                   moves   :: (DS.Set Transition),
                   initial :: NFANode,
                   final   :: (DS.Set NFANode)
                 }
          deriving (Eq,Show)
```

El nfa0 preestablecido no fue modificado.

```
nfa0 = NFA {
             sigma  = DS.fromList "ab",
             states = DS.fromList $ fmap Node [0..3],
             moves  = DS.fromList [
               Move { from = Node 0, to = Node 0, sym = 'a' },
               Move { from = Node 0, to = Node 0, sym = 'a' },
               Move { from = Node 0, to = Node 1, sym = 'a' },
               Move { from = Node 1, to = Node 2, sym = 'b' },
               Move { from = Node 2, to = Node 3, sym = 'b' }
             ],
             initial = Node 0,
             final = DS.fromList [ Node 3 ]
           }
```

Generando $\lambda-$NFAs

La generación arbitraria de nodos se emplea utilizando el constructor Node y la función getPositive que provee Test.QuickCheck.Modifiers, que provee un número positivo arbitrario x >0.

```
instance Arbitrary NFANode where
  arbitrary = Node <$> getPositive <$> arbitrary
```

Para la generación de un NFA se procede mediante los siguientes pasos:

- Se crea el nodo inicial, Nodo 0.
- Se crea una lista de caractéres de longitud arbitraria con al menos un elemento. Esto será el sigma del NFA.
- Se crea una lista de estados de longitud arbitraria que contiene al menos el nodo inicial. Esto serán los estados del NFA.
- Se selecciona al azar una de las sublistas de la lista de estados. Esto será la lista de estados finales del NFA.
- Se genera una lista de arbitraria, posiblemente vacía, usando la función frequency para tener una distribución de 1 movimiento lambda por cada 5 movimientos normales.
- Todas las listas son convertidas a Sets.
- Se genera el NFA con los valores creados.

```
instance Arbitrary NFA where
  arbitrary
    = do
        let initial = return $ Node 0
        sigmaList <- listOf1 $ choose ('a','z')
        statesList <- liftM2 (:) initial $
                              listOf ( arbitrary :: Gen NFANode )
        finalList <- elements $ List.subsequences statesList
        movesList <- listOf $
                         frequency
                           [(1, newLambda statesList),
                            (5, newMove statesList sigmaList)]
        let sigma = return $ DS.fromList sigmaList
```

```
        let states = return $ DS.fromList statesList
        let moves = return $ DS.fromList movesList
        let final = return $ DS.fromList finalList
        NFA <$> sigma <*> states <*> moves <*> initial <*> final
    where
      newLambda st
         = Lambda <$> elements st <*> elements st
      newMove st sg
         = Move <$> elements st <*> elements st <*> elements sg
```

Todas las siguientes condiciones se cumplen:

- Que el alfabeto sea no vacío sobre letras minúsculas.
- Que el conjunto de estados sea de tamaño arbitrario pero que *siempre* incluya a `(Node 0)`.
- Que tenga una cantidad arbitraria de transiciones. Todas las transiciones tienen que tener sentido: entre estados que están en el conjunto de estados y con un símbolo del alfabeto. Se desea que haya una $\lambda-$transición por cada cinco transiciones convencionales.
- El estado inicial siempre debe ser `(Node 0)`.
- El estado final debe ser un subconjunto del conjunto de estados, posiblemente vacío.

Simulador de $\lambda-$NFA

Para identificar si una transición es un movimiento normal o un movimiento lambda se hace un simple pattern matching para uno de los dos casos, en este caso isLambda, y el segundo es simplemente la negación del primero.

```
isMove, isLambda :: Transition -> Bool
isMove   = not . isLambda
isLambda t = case t of
               (Lambda _ _) -> True
               _            -> False
```

La resolución a cada función auxiliar se presenta a continuación con una breve explicación.

- Dado un $\lambda-$NFA y un estado, calcular el conjunto de estados a los cuales se puede alcanzar con *una* $\lambda-$transición.
  Esto se calcula filtrando todas las transiciones del NFA para obtener solamente los que son una lambda transición. Luego, se filtran de nuevo para obtener todos los que tienen el nodo que se pasa como parámetro como el nodo de salida de la transición. Por último, se obtiene el conjunto de todos los nodos de llegada de todos los movimientos que pasaron el filtro.

  ```
  lambdaMoves :: NFA -> NFANode -> DS.Set NFANode
  lambdaMoves nfa node = DS.map to $
                           DS.filter ((==) node . from) $
                             DS.filter isLambda $
                               moves nfa
  ```

- Dado un $\lambda-$NFA, un caracter del alfabeto y un estado, calcular el conjunto de estados a los cuales se puede alcanzar con una transición que consuma el caracter de entrada. El procedimiento es similar al anterior.
  Primero se filtran todos los movimientos del NFA para obtener los que son transiciones normales. Luego se filtran de nuevo para obtener todos los que tienen el nodo que se pasa como parámetro como el nodo de salida de la transición. Posteriormente, se filtra de nuevo para obtener todos los movimientos cuyo símbolo que consume en la transición es igual al Char pasado como parámetro. Por último, se obtiene

el conjunto de todos los nodos de llegada de todos los movimientos que pasaron el filtro.

```
normalMoves :: NFA -> Char -> NFANode -> DS.Set NFANode
normalMoves nfa c node = DS.map to $
                           DS.filter ((==) c . sym) $
                             DS.filter ((==) node . from) $
                               DS.filter isMove $
                                 moves nfa
```

- Dado un $\lambda-$NFA, un caracter del alfabeto y un estado, calcular el conjunto de estados a los cuales se puede alcanzar consumiendo el caracter de entrada. Esta es la función que debe calcular la $\lambda-$clausura del estado inicial, los desplazamientos desde ese conjunto ahora consumiendo la entrada, y luego la $\lambda-$clausura final.

  - Para la $\lambda-$clausura inicial se ejecuta la función fixSet en conjunto con la función lambdaMoves sobre el Set que contiene al nodo de salida.
  - Despues de ejecutarse esto, se calculan los nodos alcanzables desde la $\lambda-$clausura inicial consumiendo el Char que se pasa como parámetro.
  - Por último, se ejecuta el cálculo para la $\lambda-$clausura final, que es análoga a la de la $\lambda-$clausura inicial, excepto que es sobre el Set resultante de los cálculos previos.

```
destinations :: NFA -> Char -> NFANode -> DS.Set NFANode
destinations nfa c node
    = lambdaFinal . desplazamientos $ lambdaInicial
  where
    lambdaInicial
        = fixSet (lambdaMoves nfa) (DS.singleton node)
    desplazamientos
        = DS.unions . DS.toList . (DS.map normals)
    lambdaFinal
        = fixSet (lambdaMoves nfa)
    normals
        = normalMoves nfa c
```

- La función fixSet amplía un Set de elementos, aplicandole una función que es pasada como parámetro a cada elemento del Set, produciendo un Set de Sets. Luego, todos los elementos se unen entre sí para

producir el Set ampliado. Si el nuevo Set es igual a inicial, es decir, no se puede ampliar más, la función termina. Si no es así, se trata de ampliar el nuevo Set.

```
fixSet :: Ord a => (a -> DS.Set a) -> DS.Set a -> DS.Set a
fixSet f s
  = do
      let newSet = (DS.union s) .
                       DS.unions .
                         DS.toList $
                           DS.map f s
      if newSet == s
        then s
        else fixSet f newSet
```

Una vez implantadas estas funciones, se procede a definir el monad que permitirá realizar la simulación y las funciones que ejecutan dicha simulación.

El monad Simulador será el producto de la combinación del monad Error y el monad RWS. El componente Reader llevará el NFA que se está simulando, el Writer un log de los estados por los cuales se translada en cada iteración, y el State la palabra actual y los conjuntos actuales de la simulación.

Además, se define un nuevo tipo NFALog, que será proveído al Writer, que es un Seq de Sets de Nodos. Los tipos de datos NFAReject y NFARun no fueron modificados.

```
type NFALog = Seq.Seq (DS.Set NFANode)

type Simulador a
  = RWST NFA NFALog NFARun (ErrorT NFAReject IO) a

data NFAReject = Stuck (DS.Set NFANode) String
               | Reject (DS.Set NFANode)
               deriving (Show)

instance Error NFAReject

data NFARun = NFARun { w :: String, qs :: DS.Set NFANode }
            deriving (Show,Eq)
```

La función principal de este simulador será el siguiente, donde se invoca a la

función start junto con el NFA y un NFARun con la palabra pasada como parámetro y el conjunto de nodos inicial que solamente contiene al Nodo 0.

Después de la ejecución de la simulación se chequea si el valor es una de las dos posibilidades:

Left: se rechazó o se atascó, así que se imprime el error que ocurrió.

Right: se aceptó la palabra, se imprime el log de los conjuntos recorridos.

```
runNFA :: NFA -> [Char] -> IO ()
runNFA nfa word = do
                    sim <- start nfa (initialState word)
                    case sim of
                      Left err -> print err
                      Right val -> (print . snd) val
```

Las funciones auxiliares utilizadas para ejecutar la simulación son las siguientes:

- Una función para preparar el estado inicial de la simulación, con la palabra de entrada y fijando el estado actual en `(Node 0)`.

```
initialState :: String -> NFARun
initialState word
    = NFARun {w = word, qs = DS.singleton (Node 0)}
```

- Una función para determinar si en un conjunto de estados hay uno o más que sean estados finales de un NFA. Esto se logra intersectando el Set de los estados finales del NFA y el Set de nodos pasados como parámetro. Si hay al menos uno, luego el conjunto es no-vacío y la función devuelve True.

```
accepting :: NFA -> DS.Set NFANode -> Bool
accepting nfa nodes
    = not . DS.null $ DS.intersection (final nfa) nodes
```

- Una función monádica `start` que comienza la simulación a partir del estado inicial. Esta función llama a la función flow, el cual ejecuta la simulación como tal.

```
start :: NFA -> NFARun
                  -> IO (Either NFAReject (NFARun, NFALog))
start nfa ini = runErrorT $ execRWST flow nfa ini
```

- Una función monádica `flow` que completa la simulación. Esta función es la que debe operar en el monad combinado y hacer avanzar el $\lambda-$NFA consumiendo la entrada. Si detecta que debe rechazar la palabra, lanza la excepción adecuada; si logra procesar todo la entrada y aceptar, debe permitir acceso al historial de movimientos. Esto se logra mediante los siguientes pasos:

  - Se obtienen el NFA que se está simulando, la palabra actual, los nodos actuales y se registran dichos nodos en el log.
  - Si la palabra es vacía, ya no se pueden consumir caracteres y se evalúa si algunos de los nodos actuales son finales, si no, se rechaza. Se emplea un chequeo de último recurso para el caso borde donde se ejecutó un NFA con la palabra vacía pero había una lambda transición posible a un estado final.
  - Si no es vacía, se obtiene el próximo caracter, se obtienen los posibles destinos desde los nodos actuales consumiendo dicho caracter.
  - Si no hay posibles transiciones, se lanza el error que indica que se atascó el NFA.
  - En caso contrario, se guarda la nueva palabra y los nuevos nodos en el monad State y continúa la ejecución.

```
flow :: Simulador ()
flow
  = do
      nfa <- ask
      currentState <- get
      let currentWord = w currentState
      let currentNodes = qs currentState
      tell $ Seq.singleton currentNodes
      if null currentWord
        then
          unless (accepting nfa currentNodes) $
            if accepting nfa (lastResort nfa currentNodes)
              then tell . Seq.singleton $
                      lastResort nfa currentNodes
```

```
              else (throwError . Reject) currentNodes
        else (
          do
            let ([nextChar],newWord)
                  = splitAt 1 currentWord
            let dest
                  = getDestinations
                              nfa
                                nextChar
                                  currentNodes
            if DS.null dest
              then throwError $
                       Stuck currentNodes currentWord
              else (do
                     let newNFARun
                            = NFARun { w = newWord,
                                       qs = dest}
                     put newNFARun
                     flow)
          )
  where
    getDestinations nfa c nds
        = DS.unions . DS.toList $
              DS.map (destinations nfa c) nds
    lastResort nfa
        = fixSet (lambdaMoves nfa)
```

Como se observa en esta muestra de la salida de la ejecución de la simulación, los resultados coinciden con los esperados en el enunciado:

```
*Main> runNFA nfa0 "ab"
Reject (fromList [q2])

*Main> runNFA nfa0 "abb"
fromList [fromList [q0],fromList [q0,q1],fromList [q2],fromList [q3]]

*Main> runNFA nfa0 "abbb"
Stuck (fromList [q3]) "b"
```

A continuación se enuncian las dos propiedades QuickCheck:

- Todo $\lambda-$NFA que tenga un estado final en la $\lambda-$clausura de su estado inicial acepta la palabra vacía. Esto se logra usando una precondición

que indica que efectivamente es alcanzable un nodo final desde el inicial usando lambda transiciones. Posteriormente se ejecuta la simulación, y si termina correctamente se afirma que el resultado es de tipo Right.

```
prop_acceptsemptyword :: NFA -> Property
prop_acceptsemptyword nfa
  = monadicIO
  $
      do
        pre finalAlcanzableDesdeInicial
        sim <- run $ start nfa (initialState "")
        assert $ isRight sim
  where
    finalAlcanzableDesdeInicial
      = not . DS.null $
          DS.intersection lambdaClausuraIni (final nfa)
      where
        lambdaClausuraIni
          = fixSet (lambdaMoves nfa) . DS.singleton
                                              $ initial nfa
```

- Cuando un $\lambda-$NFA acepta una palabra de longitud n, el camino recorrido tiene longitud $n+1$. Esto se logra usando primero ejecutando la simulación, luego una precondición que requiere que el resultado sea de tipo Right (es decir, se aprobó la palabra utilizada) y posteriormente se afirma que la longitud del log es igual a la longitud de la palabra + 1.

```
prop_acceptancelength :: NFA -> String -> Property
prop_acceptancelength nfa w
  = monadicIO $ do
                  sim <- run $ start nfa (initialState w)
                  pre $ isRight sim
                  let (Right val) = sim
                  assert $ (Seq.length . snd) val
                                  ==
                            (1 + length w)
```

Otro Beta

La vida es dura. Todos los días hay que salir a la calle, buscando la fuerza para alcanzar otro beta y echar pa'lante.

```
data Otro a = Otro ((a -> Beta) -> Beta)
```

Y se hace más difícil, porque el `Beta` está en una chamba o en hacer llave con un convive, así hasta que te toque el quieto.

```
data Beta = Chamba (IO Beta)
          | Convive Beta Beta
          | Quieto
```

Se complica ver el `Beta`, porque `IO` esconde lo que tiene. Hay que inventarse una ahí para tener idea...

```
instance Show Beta where
   show (Chamba x)    = " chamba "
   show (Convive x y) = " convive(" ++ show x
                                    ++ ","
                                    ++ show y ++ ") "
   show Quieto        = " quieto "
```

A veces hay suerte, y uno encuentra algo que hacer. Uno llega con fuerza, hace lo que tiene que hacer, y allí sigues buscando otro Beta

```
hacer :: Otro a -> Beta
hacer otro@(Otro f) = undefined --Convive (f (\g -> _)) Quieto
```

pero es triste ver cuando simplemente es un quieto. No importa si traes fuerza, te quedas quieto.

```
quieto :: Otro a
quieto = Otro $ (\_ -> Quieto)
```

Pero hay que ser positivo. Hay que pensar que si uno encuentra un oficio, uno chambea. Sólo hay que darle a la chamba y de allí uno saca fuerza

```
chambea :: IO a -> Otro a
chambea c = Otro $ (\g -> Chamba (fmap g c))
```

y si el trabajo se complica, lo mejor es encontrar un convive para compartir la fuerza, aunque al final quedes tablas

```
convive :: Otro a -> Otro ()
convive (Otro f) = undefined
```

Para llegar lejos, es mejor cuadrar con un pana. Cada uno busca por su lado, y luego se juntan.

```
pana :: Otro a -> Otro a -> Otro a
pana o1 o2 = undefined
```

y así al final, cuando se junten los panas, hacen una vaca y se la vacilan

```
vaca :: [Beta] -> IO ()
vaca = undefined
```

Me pasaron el dato, que buscar el beta es como un perol, que con cada mano que le metes, siempre te hace echar pa'lante. Que consulte al chamo de sistemas, pa'que me muestre como hacer. Porque a esos peroles con cosas, que paso a paso avanzan, dizque los mentan "Monads". A mi no me dá el güiro, pero espero que ti si, menor.

```
instance Monad Otro where
  return x       = Otro (\f -> f x)
  (Otro f) >>= g = undefined
```