CI4251 - Programación Funcional Avanzada
Tarea 2

Erick Marrero
09-10981
<09-10981@usb.ve>

Mayo 22, 2015

Autómatas Finitos No Determinísticos

```
import Control.Monad
import Control.Monad.RWS
import Control.Monad.Error
import qualified Data.Sequence as Seq
import qualified Data.Set as DS
import Data.Char
import Data.Either
import Test.QuickCheck
import Control.Monad.Identity
```

Considere el siguiente conjunto de Tipos Abstractos de Datos diseñados para representar Autómatas Finitos No-Determinísticos con $\lambda-$Transiciones ($\lambda-$NFA).

Los estados de un $\lambda-$NFA serán representados con un `newtype`. Esto tiene el doble propósito de tener enteros diferenciados, para un desempeño aceptable, y poder disponer de una instancia `Show` separada con la cual mantener la tradición de que los estados sean denotados `q0`, `q1`, ...

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

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

Luego, representaremos las transiciones de un $\lambda-$NFA con un tipo completo que incluye alternativas para transiciones que consumen un símbolo de entrada y para $\lambda-$transiciones. Con el tipo completo, además de la conveniencia de funciones de acceso a los campos provista por la notación de registros, también podemos tener una instancia `Show` separada que muestre las transiciones de manera más atractiva.

```
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
```

Finalmente, aprovechando la librería `Data.Set` para representar conjuntos, podemos representar un $\lambda-$NFA de manera directa.

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

En esta definición:

- `sigma` es el conjunto no vacío de caracteres del alfabeto.
- `states` es el conjunto no vacío de estados; siempre debe incluir al menos al estado inicial.
- `moves` es el conjunto de transiciones.
- `initial` es el estado final que *siempre* sera `q0` (o sea `(Node 0)`).
- `final` es el conjunto de estados finales.

Con estas definiciones podemos construir expresiones que denoten $\lambda-$NFA

```
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

En primera instancia, se necesita crear nodos aleatorios y que estos no sean negativos, para ello se usa suchThat para chequear que sean positivos y crearlos de manera arbitraria.

```
instance Arbitrary NFANode where
  arbitrary = suchThat (liftM Node arbitrary) funcheck
            where funcheck (Node i) = 0 < i
```

En el caso de `NFA` queremos que el generador sólo produzca $\lambda-$NFA con estructura consistente. En este sentido, la instancia debe *garantizar*:

- Que el alfabeto sea no vacío sobre letras minúsculas. Para ello se usa *listOf1* para asegurar que no sea vacía y *elements* para seleccionar letras del abcedario.
- Que el conjunto de estados sea de tamaño arbitrario pero que *siempre* incluya a `(Node 0)`. De manera análoga para este caso, solo que el `(Node 0)` se introduce de manera manual
- 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. Para esto se seleccionan del conjunto de estados y de símbolos para generar los movimientos correctos usando la funcion *elements*, y se usa *frequency* para 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. En este caso como puede ser vacío entonces se usa *listOf*

```
instance Arbitrary NFA where
 arbitrary = do
           sigma1  <- listOf1 $ (elements ['a'..'z'])
           states1 <- listOf1 $ (arbitrary :: Gen NFANode )
           movimientos <-
               listOf1 $ funcMoves sigma1 (Node 0:states1)
           nodeFinals <- listOf $ elements $ (Node 0:states1)
```

```
          return $ NFA {
                 sigma  = DS.fromList $ sigma1,
                 states = DS.fromList $ Node 0:states1,
                 moves  = DS.fromList $ movimientos,
                 initial = Node 0,
                 final = DS.fromList $ nodeFinals
           }
       where funcMoves sig estados = do
               simbolo <- elements sig
               stateOrig <- elements estados
               stateDest <- elements estados
               frequency [
                   (5,return Move { from = stateOrig,
                             to = stateDest ,
                             sym = simbolo }  ),
                   (1, return Lambda {from = stateOrig,
                                      to = stateDest })]
```

Simulador de $\lambda-$NFA

Recordará de CI-3725 que los $\lambda-$NFA son equivalentes a los Autómatas Determinísticos, en virtud del algoritmo que simula el paso *simultáneo* por todos los estados válidos para cada transición. Le sugiero buscar el Sudkamp y sus notas de clase para reforzar el tema, aún cuando iremos desarrollando la solución paso a paso.

En primer lugar, es necesario identificar si un movimiento es convencional o es una $\lambda-$transición, así que debe proveer las funciones

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

En el algoritmo de simulación, la manera de desplazarse a partir de un estado particular consiste en considerar la $\lambda-$clausura del estado. Luego, para cada uno de los estados, determinar a su vez aquellos estados a los cuales es posible moverse consumiendo exactamente un símbolo de entrada. Finalmente, considerar la $\lambda-$clausura de esos estados, obteniéndose así los estados destino.

Para resolver ese problema, Ud. deberá implantar varias funciones auxiliares:

- Dado un $\lambda-$NFA y un estado, calcular el conjunto de estados a los cuales se puede alcanzar con *una* $\lambda-$transición. Para esta función se busca en todos los movimientos posibles las $\lambda-$transición a partir de un estado, y se obtienen los nuevos estados con movimientos gratis, y para los que no son $\lambda-$transición, se les pone una marca de `(Node -1)` para luego eliminarla.

```
lambdaMoves :: NFA -> NFANode -> DS.Set NFANode
lambdaMoves nfa node =
       DS.fromList $ filter (filNodes) $
               node:map (extracMoves node)
                        (DS.toList (moves nfa))
 where extracMoves node (Lambda f t) = if  node == f then t
                                       else Node (-1)
       extracMoves node (Move _ _ _) = Node (-1)
        filNodes (Node i) = i >=0
```

- 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. De manera análoga con esta función, solo que se obtendrán los estados a los que se llega consumiendo la entrada.

```
normalMoves :: NFA -> Char -> NFANode -> DS.Set NFANode
normalMoves nfa char node =
       DS.fromList $ filter (filNodes) $
               map (extracMoves nfa char node)
                   (DS.toList (moves nfa))
 where extracMoves nfa char node (Move f t i) =
                        if  node == f && char == i then t
                                       else Node (-1)
       extracMoves nfa char node (Lambda f t) = Node (-1)
        filNodes (Node i) = i >=0
```

- 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.

```

```

```
destinations :: NFA -> Char -> NFANode -> DS.Set NFANode
destinations nfa char node =
       DS.unions
               ( map (fixSet (lambdaMoves nfa))
               (DS.toList (DS.map (normalMoves nfa char)
                          (fixSet (lambdaMoves nfa)
                                  (DS.fromList [node])))))
```

- El cálculo de la $\lambda-$clausura es un algoritmo de la familia de Algoritmos de Punto Fijo. Para calcular el punto fijo, se construye un conjunto xs' resultante de aplicar f a cada elemento de xs. Si xs' es igual a xs, entonces se encontro el punto fijo. Si no son iguales, se calcula fixSet f xs'.

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

Una vez implantadas estas funciones, estará en posición de implantar el simulador monádico de $\lambda-$NFA poniendo en práctica monads y transformadores.

La función principal de este simulador será

```
runNFA :: NFA -> [Char] -> IO ()
runNFA nfa word = funEither $ runIdentity
         $ evalRWST (runErrorT (start >> flow ))
                                nfa (initialState word)
 where funEither (Right _ , s) = putStrLn $ show s
       funEither (Left l , _)  = putStrLn $ show l
```

que para un `nfa` particular simulará el procesamiento de la palabra de entrada `word`. El comportamiento de la función dependerá de lo que ocurra en la simulación. Para esta función, se usan los monads `Error`, `Reader`, `Writer`, `State` e `Identity`. Se uso el transformador `RWST`, y encima de este se coloca el monad `Error` para poder atrapar las excepciones que se explican más adelante.

El monad `Reader` se usará para llevar el NFA que se está simulando, el monad `Writer` para ir conservando los conjuntos de estados por los cuales vaya avanzando, el monad `State` para mantener la entrada restante y el conjunto de estados actuales, y el monad `Error` para poder

emitir las excepciones necesarias para el rechazo.

Ambos rechazos serán manejados como excepciones, así que necesitará

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

instance Error NFAReject
```

Este tipo de datos serán usados como error para:

- Si la palabra es *rechazada* porque se consumió la entrada pero el $\lambda-$NFA quedó en un estado no final, debe *imprimir* en pantalla un mensaje indicando que rechazó y en cuales estados se encontraba la simulación.
- Si la palabra es *rechazada* porque no hay transiciones posibles sobre el siguiente símbolo de entrada, debe *imprimir* en pantalla un mensaje indicando que rechazó indicando la entrada restante y el estado de estancamiento.

Se crea el tipo *NFATypeT*, para trabajar con los monads

```
type NFATypeT a =
       ErrorT NFAReject
                (RWST NFA
                        (Seq.Seq (DS.Set NFANode))
                                NFARun Identity) a
```

Necesitará un tipo de datos que se aprovechará en el Monad `State`

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

- Se necesita 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.fromList[Node 0] }
```

- Una función para determinar si en un conjunto de estados hay uno o más que sean estados finales de un NFA. Para realizar esta función, simplemente se van eliminando del conjunto de los nodos que son pasados como parámetros, los que no pertenezcan al conjunto de los estados finales, entonces solamente quedarán los que sean estados finales, y si el resultado no es vacío entonces hay uno o más que sean estados finales de ese NFA.

```
accepting :: NFA -> DS.Set NFANode -> Bool
accepting nfa nodos =
       not $ DS.null (DS.filter
                (flip DS.member (final nfa)) nodos)
```

- Una función monádica `start` que comienza la simulación a partir del estado inicial.

```
start :: NFATypeT ()
start = do
      nfa <- ask
      word <- get
      put $ word { qs = fixSet (lambdaMoves nfa)
                       (DS.fromList [Node 0])}
      tell $ Seq.singleton $
                (fixSet (lambdaMoves nfa)
                       (DS.fromList [Node 0]))
      return()
```

- 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.

```
flow :: NFATypeT ()
flow = do
     nfa <- ask
     word <- get
     if (null (w word)) then
       if (accepting nfa (qs word)) then
         put $ word { w = "",qs = DS.empty}
       else
         throwError $ Reject (qs word)
     else
```

```
      if DS.null (extracStates nfa (head (w word))
                                (qs word) DS.empty) then
          throwError $ Stuck (qs word) (w word)
      else
          do
            put $ word { w = tail (w word),
              qs = extracStates nfa (head (w word))
                                    (qs word) DS.empty}
            tell $ Seq.singleton
                $ extracStates nfa (head (w word))
                                    (qs word) DS.empty
            flow

 where extracStates n p e s = if (DS.null e) then s
              else extracStates n p
                      (DS.fromList (drop 1 (DS.toList e)))
                      (DS.union s
                              (destinations n p
                                  ((head (DS.toList e)))))
```

Una vez que su simulación esté operando correctamente, escriba dos propiedades QuickCheck y aproveche la instancia `Arbitrary` para comprobar:

- Todo $\lambda-$NFA que tenga un estado final en la $\lambda-$clausura de su estado inicial acepta la palabra vacía.

```
prop_acceptsemptyword :: NFA -> Property
prop_acceptsemptyword nfa =
 (DS.empty /= (DS.intersection
                      (fixSet (lambdaMoves nfa)
                              (DS.fromList [Node 0]))
                      (final nfa))) ==>
                               runNFAEmptyWord nfa ""
 where runNFAEmptyWord nfa word =
               funEither $ runIdentity $ evalRWST
                      (runErrorT (start >> flow ))
                              nfa (initialState word)
       funEither (Right _ , _) = True
       funEither (Left _ , _) = False
```

- Cuando un $\lambda-$NFA acepta una palabra de longitud n, el camino recorrido tiene longitud $n+1$. Para probar esta propiedad, se tiene que aceptar la palabra para que se cumpla el antecedente, para ello se ejecuta el siguiente comando:

```
ghci> quickCheckWith (stdArgs { maxDiscardRatio = 500})
  prop_acceptancelenght
```

```
prop_acceptancelength :: NFA -> String -> Property
prop_acceptancelength nfa w = let (x,tam) =
                                        runNFAWord nfa w
                              in x ==> length w +1 == tam
 where runNFAWord nfa word = funEither1 $ runIdentity
         $ evalRWST (runErrorT (start >> flow ))
                              nfa (initialState word)
        funEither1 (Right _ , l) = (True,Seq.length l)
        funEither1 (Left _ , _) = (False, 0)
```

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 f) = Convive (f (\_ -> Quieto)) 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. Se usa `fmap` para aplicar la funcion y luego dejarlo en `IO`

```
chambea :: IO a -> Otro a
chambea a = Otro (\f -> Chamba (fmap f a))
```

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 a = Otro (\_ -> Convive (hacer a) Quieto)
```

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 (Otro x) (Otro y) =
               Otro(\f -> Convive (x f) (y (\_ -> x f)))
```

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

```
vaca :: [Beta] -> IO ()
vaca []                = return ()
vaca (Chamba x   :xs) = do
                          x' <- x
                          vaca (x':xs)
vaca (Convive x y:xs) = vaca (xs++[x,y])
vaca (Quieto     :xs) = vaca xs
```

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 = Otro (\_ -> f (hacer.g))
```

El resultado final es el siguiente:

```
ghci> quedo cartel
http://tinyurl.com/2fcpre6
```