CI4251 - Programación Funcional Avanzada
Entrega Tarea 1

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

Mayo 01, 2015

1. Librerías Utilizadas

```
import Control.Applicative ((<$>),(<*>), pure)
import Control.Monad ((>=>))
import Data.List
import Data.Functor
import Data.Maybe
import Data.Monoid
import Data.Foldable (foldMap)
import Data.Tree
```

Las librerías ya suministradas no fueron alteradas. Solamente fueron removidas los imports referentes a librerías de elaboración de gráficos por problemas de instalación y no eran requisitos para la correcta ejecución del ejercicio. Adicionalmente se importaron tres librerías, Data.Maybe, Control.Applicative y Control.Monad para la elaboración del segundo ejercicio y la prueba del tercer ejercicio.

2. Machine Learning

Se empieza trabajando con estos datos ya suministrados por el enunciado.

```
data Sample a = Sample { x :: [a], y :: a }
     deriving (Show)

data Hypothesis a = Hypothesis { c :: [a] }
     deriving (Show)
```

Adicionalmente, se tienen estos valores por defecto que se emplearán a lo largo de la resolución del problema. El contenido de *training* no se muestra en este documento por razones de legibilidad, pero al cargarlo en ghci se evidencia como es una lista con varios Samples ya predeterminados.

```
alpha :: Double
alpha = 0.03

epsilon :: Double
epsilon = 0.0000001

guess :: Hypothesis Double
guess = Hypothesis { c = [0.0, 0.0, 0.0] }

training :: [Sample Double]
```

2.1. Comparar en punto flotante

Para esta sección la función es bastante sencilla. Para ver si la diferencia entre dos numeros es ϵ-despreciable se necesita simplemente restar el mayor de los dos menos el menor, y consecuentemente verificar que ese resultado sea menor al ϵ preestablecido en la sección anterior.

```
veryClose :: Double -> Double -> Bool
veryClose v0 v1 = max v0 v1 - min v0 v1 < epsilon
```

2.2. Congruencia dimensional

En esta sección, ateniendose al requisito de que la función sea *point-free* y que cada Sample en la lista tenga un 1 al comienzo de su lista de X, se utiliza la función map para alterar cada Sample junto con una función anónima que aprovecha la funcionalidad de Record Update junto con un cons para agregar el 1 de forma rápida y eficiente.

```
addOnes :: [Sample Double] -> [Sample Double]
addOnes = map (\s -> s {x = 1 : x s})
```

2.3. Evaluando Hipótesis

En esta sección se desarrollan dos funciones, `theta` y `cost`.

Para la función `theta`, como se puede asumir que ambos vectores tienen las mismas dimensiones, se pueden usar las funciones de la familia zip, en particular zipWith, sin riesgo de perder algún dato.

La función `theta` es equivalente al producto punto, es decir, multiplicar cada x_i por cada h_i y cada multiplicación es el nuevo θ_i, y posteriormente se suman todos los θ_i para obtener el valor final θ, es análogo a usar las funciones predefinidas de Haskell de *zipWith (*)* y posteriormente *sum* de dicho zip.

```
theta :: Hypothesis Double -> Sample Double -> Double
theta h s = sum $ zipWith (*) (x s) (c h)
```

Para la función `cost`, como se se exige que el resultado sea calculado en una sola pasada y sin asumir una longitud para la lista de Samples, se emplea el uso de un *fold* que utiliza una función auxiliar que permite realizar la sumatoria necesaria. Adicionalmente, como se hace énfasis en una sola pasada, y se necesita la longitud de la lista de Samples, utilizar la

función predefinida *length* no es viable, ya que esto involucraría otra pasada por la lista. Por esto, la función auxiliar tiene como acumulador una tupla, con el primer elemento siendo el resultado de la sumatoria y el segundo elemento un contador que al final del *fold* será igual a la longitud de la lista de Samples.

```
cost :: Hypothesis Double -> [Sample Double] -> Double
cost h ss
  = fst res / (2 * snd res)
  where
    aux (acum,len) next
     = (acum + (theta h next - y next)^2, len+1)
    res
     = foldl' aux (0,0) ss
```

2.4. Bajando por el gradiente

En esta sección se desarrollan dos funciones, `descend` y `gd`.

Para la función `descend`, que debe ejecutar un cálculo para reemplazar cada θ_i por un θ'_i, es una situación ideal para utilizar la función preestablecida *map* de Haskell. Como los cálculos requieren que se sepa el índice i del θ_i que se esta reemplazando primero se ejecuta un *zip* con la lista [0..] y la lista de thetas de la hipótesis, para así tener acceso rápido a dicho número.

La función que ejecutará el *map* es una función auxiliar `newtheta` que toma una tupla (θ,i) que devuelve el nuevo θ' según la fórmula matemática. Dicha función hace una llamada a otra función auxiliar la cual es un *fold* que va generando una tupla que al final contiene el resultado de la fórmula matemática y la longitud de la lista de Samples.

```
descend :: Double -> Hypothesis Double -> [Sample Double]
        -> Hypothesis Double
descend alpha h ss
  = h { c = map newtheta $ zip (c h) [0..] }
  where
    newtheta (theta,index)
      = theta - (alpha/snd (res index))*(fst (res index))
    res index
      = foldl' (aux index) (0,0) ss
    aux index (acum,len) next
      = (acum + (theta h next - y next)*((x next) !! index),
                                                    len+1)
```

Para la función `gd`, se implementa mediante un *unfoldr* que utiliza una

función auxiliar `expandir` que va devolviendo una 3-tupla

(iteración, próxima hipótesis, costo de dicha hipótesis)

Dicho *unfoldr* termina cuando la diferencia entre los costos entre una hipótesis y la siguiente es ϵ-despreciable. Adicionalmente, se agregan unos al principio de cada vector X de cada Sample utilizando la función `addOnes`.

```
gd :: Double -> Hypothesis Double -> [Sample Double]
   -> [(Integer,Hypothesis Double,Double)]
gd alpha h ss
  = unfoldr expandir (0,h,cost h ssOnes)
  where
    expandir (iter,hipotesis,costo)
      = case veryClose costo (nextCosto hipotesis) of
          True  -> Nothing
          False -> Just ( ( iter,
                            hipotesis,
                            costo
                          ),
                          ( iter+1,
                            nextHipotesis hipotesis,
                            nextCosto hipotesis
                          )
                        )
    nextHipotesis oldHipotesis
      = descend alpha oldHipotesis ssOnes
    nextCosto oldHipotesis
      = cost (nextHipotesis oldHipotesis) ssOnes
    ssOnes
      = addOnes ss
```

3. Monoids

En esta sección se debe construir una instancia Monoid polimórfica para cualquier tipo comparable tal que se extraiga de cualquier Foldable conteniendo elementos de dicho tipo el máximo valor asignado. Como puede ser para cualquier tipo de datos, Num por ejemplo, se emplea el método de envolverlos con el tipo de data `Max a`.

Para proporcionar el requisito de que el Monoid opere con seguridad se utiliza el monad *Maybe* para envolver el resultado, de esta forma si el Foldable se encuentra vacío simplemente se devuelve un Nothing. Si hay un resultado se devuelve envuelto en un Just.

Como se indica en la instancia del Monoid, el tipo polimórfico debe ser un *Ord a*, asi satisfaciendo el requerimiento de que sea sobre cualquier tipo comparable.

La definición de *mempty* es directo, es simplemente Max Nothing, ya que si es vacío devuelve nada, como indican los ejemplos proporcionados en el enunciado. La definición de *mappend* emplea el uso de los operadores <$>y <*>de Control.Applicative y la función *maybe* de Data.Maybe para obtener el resultado con la función *max*.

La lógica que se sigue es la siguiente:

- La función max recibe dos parámetros del tipo a.
- Los elementos del Max estan en el contexto Maybe.
- Luego, se ejecuta *max <$>param1 <*>param2*, lo cual saca a param1 y param2 de su contexto Maybe, los pasa como parámetros al *max* y los vuelve a poner en su contexto Maybe.
- Para evitar cláusulas de pattern matching se emplean las funciones:

  maybe x pure y maybe y pure x

- Esto impide si se da la situación de:
  - x = Nothing
  - y = Just valor

  Que el Nothing absorba el Just valor, ya que dado lo anterior se tendría Just valor <*>Just valor, y el valor es preservado.

```
data Max a = Max { getMax :: Maybe a }
     deriving (Eq, Ord, Read, Show)

instance (Ord a) => Monoid (Max a) where
  mempty
    = Max Nothing
  Max x `mappend` Max y
    = Max $ max <$> maybe x pure y <*> maybe y pure x
```

4. Zippers

En esta sección se implementará un Zipper para el tipo de dato suministrado por el enunciado que se presenta a continuación.

```
data Filesystem a = File a | Directory a [Filesystem a]
  deriving (Show)
```

Empleando el método de traducir `Filesystem` a su tipo algebráico y consecuentemente aplicando la derivada parcial, se obtiene que el tipo de dato necesario para representar las "migajas"del recorrido es el siguiente:

```
data Crumbs a = Crumbs a [Filesystem a] [Filesystem a]
  deriving (Show)
```

La interpretación que se da a este tipo de dato es:

Crumbs x l r

Donde:

- x = Nombre del directorio padre
- l = Elementos a la izquierda de elemento enfocado en directorio actual
- r = Elementos a la derecha de elemento enfocado en directorio actual

Posterior a esto se declara el tipo `Breadcrumbs` que no es más que una lista de `Crumbs` y el tipo `Zipper`, que es la tupla que representa el `Filesystem` actualmente en foco y la lista del recorrido hasta ahora realizado.

```
type Breadcrumbs a = [Crumbs a]

type Zipper a = (Filesystem a, Breadcrumbs a)
```

Para la función `goDown` se tienen dos casos:

- Si se está enfocado en un `File`, luego no se puede bajar.
- Si se está enfocado en un `Directory`, luego se baja, agregando un Crumb con el nombre del Directorio, una lista vacía para elementos a la izquierda, una lista de todos los elementos en el directorio en la lista para elementos a la derecha y por último enfocando el primer elemento del directorio al cual se bajó.

Como se requiere que el `Zipper` sea seguro, tratar de bajar por un archivo devuelve Nothing, bajar por un directorio devuelve Just nuevoZipper.

```
goDown :: Zipper a -> Maybe (Zipper a)
goDown (File fl, bs)
  = Nothing
goDown (Directory d (f:fs), bs)
  = Just (f, Crumbs d [] fs : bs)
```

Para la función `goRight` se tienen dos casos:

- Si no hay más elementos a la derecha, no se puede mover.
- Si hay elementos a la derecha, luego el Crumb más reciente es modificado por uno con la lista de elementos de cada lado actualizados para representar el cambio y se enfoca en el elemento inmediatamente a la derecha.

Como se requiere que el `Zipper` sea seguro, tratar de moverse a la derecha cuando esto no es posible devuelve Nothing, cuando si es posible devuelve Just nuevoZipper.

```
goRight :: Zipper a -> Maybe (Zipper a)
goRight (fs, Crumbs _ _ []:bs)
  = Nothing
goRight (fs, Crumbs d ls (r:rs):bs)
  = Just (r, Crumbs d (fs:ls) rs:bs)
```

Para la función `goLeft` se tienen dos casos:

- Si no hay más elementos a la izquierda, no se puede mover.
- Si hay elementos a la izquierda, luego el Crumb más reciente es modificado por uno con la lista de elementos de cada lado actualizados para representar el cambio y se enfoca en el elemento inmediatamente a la izquierda.

Como se requiere que el `Zipper` sea seguro, tratar de moverse a la izquierda cuando esto no es posible devuelve Nothing, cuando si es posible devuelve Just nuevoZipper.

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (fs, Crumbs _ [] _:bs)
  = Nothing
goLeft (fs, Crumbs d (l:ls) rs:bs)
  = Just (l, Crumbs d ls (fs:rs):bs)
```

Para la función `goBack` se tienen dos casos:

- No hay más `Crumbs`, es decir, se esta en el directorio raíz, luego no se puede subir más.
- Si hay `Crumbs` restantes, es decir, hay al menos un directorio que contiene al `Filesystem` actualmente en foco. Luego, se reconstruye el directorio padre con su nombre almacenado en el `Crumb` y reconstruyendo la lista de los `Filesystem` contenidos en él concatenando la lista de elementos a la izquierda del `Filesystem` en foco, el `Filesystem` en foco y la lista de elementos a la derecha del `Filesystem` en foco. Adicionalmente, el `Crumb` donde estaba almacenado esta información es desechado y se guardan las que venían antes que él.

Como se requiere que el `Zipper` sea seguro, tratar de subir cuando esto no es posible devuelve Nothing, cuando si es posible devuelve Just nuevoZipper.

```
goBack :: Zipper a -> Maybe (Zipper a)
goBack (fs, [])
  = Nothing
goBack (fs, Crumbs d ls rs:bs)
  = Just (Directory d $ reverse ls ++ [fs] ++ rs,bs)
```

La función `tothetop` es una aplicación recursiva de goBack, para subir por los directorios hasta llegar al directorio raíz y se devuelve el `Zipper` resultante. Cuando la lista de `Crumbs` está vacía ya se ha llegado al directorio raíz. Si hay por lo menos uno, entonces se invoca `goBack` y se vuelve a aplicar `tothetop` sobre este nuevo `Zipper`.

```
tothetop :: Zipper a -> Zipper a
tothetop (fs, [])
  = (fs, [])
tothetop z
  = tothetop $ fromJust $ goBack z
```

La función `modify` aplica una función al `Filesystem` actualmente en foco y devuelve un Zipper con dicho `Filesystem` modificado. El elemento modificado de un `File` es el único que posee, el elemento modificado de un `Directory` es el primero.

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Directory d fs, bs)
  = (Directory (f d) fs, bs)
modify f (File fl, bs)
  = (File (f fl), bs)
```

La función `focus` recibe como entrada un `Filesystem` y devuelve un nuevo `Zipper` con dicho `Filesystem` en foco y sin `Crumbs`.

```
focus :: Filesystem a -> Zipper a
focus fs = (fs, [])
```

La función `defocus` recibe como su entrada un `Zipper` y devuelve el `Filesystem` en foco y se descartan los `Crumbs` si todavía hay.

```
defocus :: Zipper a -> Filesystem a
defocus (fs, _) = fs
```

Para demostrar la ejecución de estos comandos, se parte de una estructura hecha de un directorio que contiene otros directorios y archivos. Se navegará hasta el archivo "bash.sh"para cambiar su nombre a "newbash.sh". Luego se irá hasta el directorio padre y se desenfocará para ver si se preserva el cambio. Por legibilidad se utiliza el operador `(>=>)` de Control.Monad para concatenar fácilmente las operaciones de movimiento.

```
folders :: Filesystem String
folders
  = Directory "/root" [
                       File "img01.jpeg",
                       File "img02.jpeg",
                       Directory "/scripts" [
                                             File "bash.sh",
                                             File "test.sh",
                                             File "hax.sh"
                                            ],
                       File "to-do-list.txt"
                      ]

zipper :: Zipper String
zipper = focus folders
```

Viendo el output de ghci, se evidencia que el cambio se realizó de forma exitosa:

```
*Main> folders
```

```
Directory "/root" [File "img01.jpeg",
                   File "img02.jpeg",
                   Directory "/scripts" [File "bash.sh",
                                         File "test.sh",
                                         File "hax.sh"],
                   File "to-do-list.txt"]

*Main> defocus $ tothetop $ modify (\_ -> "newbash.sh") $
         fromJust $
           (goDown >=> goRight >=> goRight >=>
              goLeft >=> goRight >=> goDown) zipper

Directory "/root" [File "img01.jpeg",
                   File "img02.jpeg",
                   Directory "/scripts" [File "newbash.sh",
                                         File "test.sh",
                                         File "hax.sh"],
                   File "to-do-list.txt"]
```