Trabajo Práctico # 4

Programación Funcional, Universidad Nacional de Quilmes

8 de mayo de 2018

Aclaraciones:

- *Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados. No se saltee ejercicios sin consultar antes a un docente.*
- *Recuerde que puede aprovechar en todo momento las funciones que ha definido, tanto las de esta misma práctica como las de prácticas anteriores.*
- *Pruebe todas sus implementaciones, al menos en una consola interactiva.*
- *Es sumamente aconsejable resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en clase, dado que los exámenes de la materia evaluación principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*

1. Map

1.1.

Definir la función map para los siguientes tipos algebraicos:

1. `data [a] = [] | a : [a]`
2. `data Tree a = EmptyT | NodeT a (Tree a) (Tree a)`
3. `data NonEmptyList a = Unit a | NECons a (NonEmptyList a)`
4. `data AppendList a = Nil | Unit a | Append (AppendList a) (AppendList a)`
5. `data Maybe a = Nothing | Just a`
6. `data T a = A a | B (T a) | C (T a) (T a)`
7. `data LTree a = L [a] | B a (M a) (M a)`
8. `data Either b a = Left b | Right a`
9. `data MTree a = L (Maybe a) | B a (MTree a) (MTree a)`
10. (Desafío) `data GenTree a = GNode a [GenTree a]`

1.2.

Con la definición de map dada para el tipo `T` del ejercicio anterior (llamémosla `mapX`), demostrar:

1. `mapX id = id`
2. `mapX f . mapX g = mapX (f.g)`
   Llamada propiedad de fusión

1.3.

Definir las siguientes funciones sobre los tipos algebraicos del punto 1, donde `f` es reemplazado por cada uno de dichos tipos:

1. `find :: (a -> Bool) -> f a -> Maybe a`
2. `any, all :: (a -> Bool) -> f a -> Bool`
3. `partition :: (a -> Bool) -> f a -> ([a], [a])`

2. Fold

2.1. Definición

Definir el fold de los siguientes tipos algebraicos:

1. `data [a] = [] | a:[a]`
2. `data Tree a = EmptyT | NodeT a (Tree a) (Tree a)`
3. `data NonEmptyList a = Unit a | NECons a (NonEmptyList a)`
4. `data AppendList a = Nil | Unit a | Append (AppendList a) (AppendList a)`
5. `data IntList = IntNil | IntCons Int IntList`
6. ```
   data Exp = Const Int
            | Var String
            | Sum Exp Exp
            | Prod Exp Exp
   ```
7. `data Maybe a = Nothing | Just a`
8. `data Either a b = Left a | Right b`
9. `data Nat = Zero | Succ Nat`
10. `data T a = A a | B (T a) | C (T a) (T a)`
11. `data LTree a = L [a] | B a (M a) (M a)`
12. `data MTree a = L (Maybe a) | B a (MTree a) (MTree a)`
13. `data Exp = Var String | Const Int | Suma Exp Exp`

2.2.

Definir para listas:

1. `foldr1 :: (a -> a -> a) -> [a] -> a`
2. `recr :: (a -> [a] -> b -> b) -> b -> [a] -> b`

2.3.

Definir todas las funciones ya definidas sobre listas `foldr`, `foldr1` o `recr` (dependiendo del caso), y sobre árboles binarios con foldT. Tener en cuenta que en algunos casos pueden ser difíciles, o que no se puedan realizar con recursión estructural.
   ```

2.4.

Definir las siguientes funciones:

1. `scanr :: (a -> b -> b) -> b -> [a] -> [b]`
2. `foldl :: (a -> b -> a) -> a -> [b] -> a`
3. `foldl1 :: (a -> a -> a) -> [a] -> a`
4. `scanl :: (a -> b -> a) -> a -> [b] -> [a]`
5. `unfoldr :: (b -> Maybe (a, b)) -> b -> [a]`

2.5. Desafío

Definir `recT` (recursión primitiva sobre árboles binarios) y definir `leaves :: Tree a -> [a]` usando `recT`.

3. Foldr vs Foldl

1. Ejecutar

   ```
   >> foldl (-) 0 [1..5]
   >> foldr (-) 0 [1..5]
   ```

   ¿Por qué el resultado no es el mismo?

2. Explicar la diferencia entre `foldr` y `foldl`

3. Ejecutar las siguientes funciones con algún ejemplo que demuestre la naturaleza de foldr y foldl:

   ```
   showFoldrMagic :: (Show a) => String -> String -> [a] -> String
   showFoldrMagic f z = foldr (\x r -> concat ["(",f," ",show x," ",r,")"]) z

   showFoldlMagic :: (Show a) => String -> String -> [a] -> String
   showFoldlMagic f z = foldl (\r x -> concat ["(",f," ",r," ",show x,")"]) z
   ```

4. (Desafío) Definir `foldr` usando `foldl`, y viceversa. ¿Se mantienen las mismas propiedades con dichas definiciones?

4. Demostraciones

Demostrar las siguientes equivalencias y propiedades, comparando la implementación sin `fold` para todas las funciones):

1. `concat = foldr (++) []`
2. `map f = foldr ((:) . f) []`
3. `foldr f z (xs ++ ys) = foldr f (foldr f z ys) xs`
4. `foldr f z . foldr (:) [] = foldr f z`
5. `foldT NodeT EmptyT = id`
6. `filter p = (\p -> foldr (\x -> if p x then (x:) else id) [])`

7. `(+1) . sum = foldr (+) 1`
8. `(n*) . sum = foldr ((+) . (n*)) 0`
9. (Desafío) Si `h (f x y) = g x (h y)`
   entonces `h . foldr f z = foldr g (h z)`

5. Árboles Generales

Definir las dos versiones de fold para `data GenTree a = GNode a [GenTree a]`, y usarlas para dar dos definiciones a cada una de las siguientes funciones sobre árboles generales:

1. `sumGT :: GenTree Int -> Int`
2. `sizeGT :: GenTree a -> Int`
3. `heightGT :: GenTree a -> Int`
4. `anyGT :: (a -> Bool) -> GenTree a -> Bool`
5. `allGT :: (a -> Bool) -> GenTree a -> Bool`
6. `countByGT :: Eq a => (a -> Bool) -> GenTree a -> Int`
7. `mapGT :: (a -> b) -> GenTree a -> GenTree b`
8. `mirrorGT :: GenTree a -> GenTree a`
9. `toListGT :: GenTree a -> [a]`
10. (Desafío) `levelNGT ::  GenTree a -> Int -> [a]`
11. (Desafío) `allPathsGT :: GenTree a -> [[a]]`
12. (Desafío) `longestPathGT :: GenTree a -> [a]`
13. (Desafío) `listPerLevelGT :: GenTree a -> [[a]]`

5.1. Desafío

Definir `recGT` (recursión primitiva sobre GenTree) y definir `leavesGT :: GenTree a -> [a]` usando `GT`