Programación Funcional

Ejercicios de Práctica Nro. 12

Esquemas de funciones II

Aclaraciones:

- Los ejercicios siguen un orden de complejidad creciente, y cada uno puede servir a los siguientes. No se recomienda saltear ejercicios sin consultar antes a un docente.
- Recordar que se pueden aprovechar en todo momento las funciones ya definidas, tanto las de esta misma práctica como las de prácticas anteriores.
- Probar todas las 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 evalúan principalmente este aspecto. Para utilizando formas alternativas al resolver los ejercicios consultar a los docentes.

Ejercicio 1) Dada la definición de **`ExpA`**:

```
data ExpA = Cte Int
          | Suma ExpA ExpA
          | Prod ExpA ExpA
```

a. Dar el tipo y definir **`foldExpA`**, que expresa el esquema de recursión estructural para la estructura **`ExpA`**.

b. Resolver las siguientes funciones utilizando **`foldExpA`**:
   i. **`cantidadDeCeros :: ExpA -> Int`**, que describe la cantidad de ceros explícitos en la expresión dada.
   ii. **`noTieneNegativosExplicitosExpA :: ExpA -> Bool`**, que describe si la expresión dada no tiene números negativos de manera explícita.
   iii. **`simplificarExpA' :: ExpA -> ExpA`**, que describe una expresión con el mismo significado que la dada, pero que no tiene sumas del número 0 ni multiplicaciones por 1 o por 0. La resolución debe ser exclusivamente *simbólica*.
   iv. **`evalExpA' :: ExpA -> Int`**, que describe el número que resulta de evaluar la cuenta representada por la expresión aritmética dada.
   v. **`showExpA :: ExpA -> String`**, que describe el string sin espacios y con paréntesis correspondiente a la expresión dada.

c. Demostrar que **`evalExpA'`** es equivalente a **`evalExpA`** (ejercicio 6.a.i de la práctica 8).

d. Dar el tipo y definir **`recExpA`**, que expresa el esquema de recursión primitiva para la estructura **`ExpA`**.

e. Resolver las siguientes funciones utilizando **`foldExpA`**:

i. **`cantDeSumaCeros :: ExpA -> Int`**, que describe la cantidad de constructores de suma con al menos uno de sus hijos constante cero.
ii. **`cantDeProdUnos :: ExpA -> Int`**, que describe la cantidad de constructores de producto con al menos uno de sus hijos constante uno.

Ejercicio 2) Dada la definición de **`EA`**:

```
data EA = Const Int | BOp BinOp EA EA
data BinOp = Sum | Mul
```

a. Dar el tipo y definir **`foldEA`**, que expresa el esquema de recursión estructural para la estructura **`EA`**.

b. Resolver las siguientes funciones utilizando **`foldEA`**:
   i. **`noTieneNegativosExplicitosEA :: EA -> Bool`**, que describe si la expresión dada no tiene números negativos de manera explícita.
   ii. **`simplificarEA' :: EA -> EA`**, que describe una expresión con el mismo significado que la dada, pero que no tiene sumas del número 0 ni multiplicaciones por 1 o por 0. La resolución debe ser exclusivamente *simbólica*.
   iii. **`evalEA' :: EA -> Int`**, que describe el número que resulta de evaluar la cuenta representada por la expresión aritmética dada.
   iv. **`showEA :: EA -> String`**, que describe el string sin espacios y con paréntesis correspondiente a la expresión dada.
   v. **`ea2ExpA' :: EA -> ExpA`**, que describe una expresión aritmética representada con el tipo ExpA, cuyo significado es el mismo que la dada.
   vi. **`ea2Arbol' :: EA -> ABTree BinOp Int`**, que describe la representación como elemento del tipo **`ABTree BinOp Int`** de la expresión aritmética dada.

c. Demostrar que **`evalEA'`** es equivalente a **`evalEA`** (ejercicio 1.a.i de la práctica 9).

Ejercicio 3) Dada la definición de **`Tree`**:

```
data Tree a = EmptyT | NodeT a (Tree a) (Tree a)
```

a. Dar el tipo y definir la función **`foldT`**, que expresa el esquema de recursión estructural para la estructura **`Tree`**.

b. Definir las siguientes funciones utilizando **`foldT`**:

i. `mapT :: (a -> b) -> Tree a -> Tree b`
ii. `sumT :: Tree Int -> Int`
iii. `sizeT :: Tree a -> Int`
iv. `heightT :: Tree a -> Int`
v. `preOrder :: Tree a -> [a]`
vi. `inOrder :: Tree a -> [a]`
vii. `postOrder :: Tree a -> [a]`
viii. `mirrorT :: Tree a -> Tree a`
ix. `countByT :: (a -> Bool) -> Tree a -> Int`
x. `partitionT :: (a -> Bool) -> Tree a -> ([a], [a])`
xi. `zipWithT :: (a->b->c) -> Tree a -> Tree b -> Tree c`
xii. `caminoMasLargo :: Tree a -> [a]`
xiii. `todosLosCaminos :: Tree a -> [[a]]`
xiv. `todosLosNiveles :: Tree a -> [[a]]`
xv. `nivelN :: Tree a -> Int -> [a]`

c. Dar el tipo y definir la función **`recT`**, que expresa el esquema de recursión primitiva para la estructura **`Tree`**.

d. Definir las siguientes funciones utilizando **`recT`**:
   i. **`insertT :: a -> Tree a -> Tree a`**, que describe el árbol resultante de insertar el elemento dado en el árbol dado, teniendo en cuenta invariantes de BST.
   ii. **`caminoHasta :: Eq a => a -> Tree a -> [a]`**, que describe el camino hasta el elemento dado en el árbol dado.
   Precondición: existe el elemento en el árbol.

e. Demostrar las siguientes propiedades utilizando las definiciones anteriores:
   i. para todo f. `sizeT . mapT f = sizeT`
   ii. para todo f. para todo g. `mapT f . mapT g = mapT (f . g)`
   iii. `foldT EmptyT NodeT = id`

Ejercicio 4) Dada la siguiente definición:

```
type Record a b = [(a,b)]
```

donde la idea de este tipo es representar una fila de una base de datos (el valor de tipo a es el nombre del campo, y el valor de tipo b es el valor de ese campo).

Con esto, puede definirse el tipo:

```
type Table a b = [ Record a b ]
```

donde se entiende que una tabla de una base de datos está compuesta por muchos registros, y se espera que todos compartan los mismos “campos” (o sea, los valores de tipo a en cada registro).

Definir las siguientes funciones sin utilizar recursión estructural explícita:

a. **`select :: (Record a b -> Bool) -> Table a b -> Table a b`**, que a partir de la lista de registros dada describe la lista de los registros que cumplen con la condición dada.
b. **`project :: (a -> Bool) -> Table a b -> Table a b`**, que a partir de la lista de registros dada describe la lista de registros solo con los campos que cumplen la condición dada.
c. **`conjunct :: (a -> Bool) -> (a -> Bool) -> a -> Bool`**, que describe el predicado que da True solo cuando los dos predicados dados lo hacen.
d. **`crossWith :: (a -> b -> c) -> [a] -> [b] -> [c]`**, que describe el resultado de aplicar una función a cada elemento del producto cartesiano de las dos listas de registros dadas.
e. **`product :: Table a b -> Table a b -> Table a b`**, que describe la lista de registros resultante del producto cartesiano combinado de las dos listas de registros dadas. Es decir, la unión de los campos en los registros del producto cartesiano.
f. **`similar :: Record a b -> Record a b`**, que describe el registro resultante de descartar datos cuyos campos sean iguales (o sea, el mismo dato asociado al mismo campo).

Ejercicio 5) Una expresión del cálculo relacional de tuplas permite expresar consultas sobre un modelo relacional con los siguientes componentes:

```
data Query a b
   = Table [Record a b]            -- Table (Table a b)
   | Product (Query a b) (Query a b)
   | Projection (a -> Bool) (Query a b)
   | Selection (Record a b -> Bool) (Query a b)
```

Este modelo es utilizado para estudiar y manipular las consultas hechas a motores de bases de datos. En este contexto vemos a cada registro como una fila y a cada elemento como un par columna-valor. Por ejemplo:

```
Projection
   (/= "age")
   (Selection
       (\r -> any (\(c,v)-> c == "name"
                         && v == "Edward Snowden") r)
       (Table [ [("name", "Edward Snowden"),("age", "29")]
                [("name", "Jason Bourne"),  ("age", "40")] ]))
```

a. Dar el tipo y definir la función **`foldQ`**, que expresa el esquema de recursión estructural para la estructura **`Query`**.
a. Definir las siguientes funciones sin utilizar recursión explícita:
    i. **`tables :: Query a b -> [Table a b]`**, que describe la lista de todas las tablas involucradas en la query dada.
    ii. **`execute :: Query a b -> Table a b`**, que describe el resultado de ejecutar la query dada.

iii. **`compact :: Query a b -> Query a b`**, que describe la query resultante de compactar las selecciones y proyecciones consecutivas en la query dada.

b. Demostrar la siguiente propiedad:

```
execute . compact = execute
```

Ejercicio 6) Dadas las siguientes definiciones para representar mapas con diferentes puntos de interés que pueden presentar objetos:

```
data Dir = Left | Right | Straight
data Mapa a = Cofre [a]
            | Nada (Mapa a)
            | Bifurcacion [a] (Mapa a) (Mapa a)
```

a. Dar el tipo y definir **`foldM`** y **`recM`**, que expresan los esquemas de recursión estructural y primitiva, respectivamente, para la estructura **`Mapa`**.

b. Definir las siguientes funciones sin utilizar recursión explícita:

i. **`objects :: Mapa a -> [a]`**, que describe la lista de todos los objetos presentes en el mapa dado.

ii. **`mapM :: (a -> b) -> Mapa a -> Mapa b`**, que transforma los objetos del mapa dado aplicando la función dada.

iii. **`has :: (a -> Bool) -> Mapa a -> Bool`**, que indica si existe algún objeto que cumpla con la condición dada en el mapa dado.

iv. **`hasObjectAt :: (a->Bool) -> Mapa a -> [Dir] -> Bool`**, que indica si un objeto al final del camino dado cumple con la condición dada en el mapa dado.

v. **`longestPath :: Mapa a -> [Dir]`**, que describe el camino más largo en el mapa dado.

vi. **`objectsOfLongestPath :: Mapa a -> [a]`**, que describe la lista con los objetos presentes en el camino más largo del mapa dado.

vii. **`allPaths :: Mapa a -> [[Dir]]`**, que describe la lista con todos los caminos del mapa dado.

viii. **`objectsPerLevel :: Mapa a -> [[a]]`**, que describe la lista con todos los objetos por niveles del mapa dado.

c. Demostrar la siguiente propiedad:

para todo **`x`**. **`has (==x) = any (elem x) . objectsPerLevel`**

Ejercicio 7) Dada la siguiente definición para representar árboles generales:

```
data GTree a = GNode a [GTree a]
```

a. Dar tipo y definir las tres versiones de **`foldGT`** y **`recGT`**, que expresan los esquemas de recursión estructural y primitiva, respectivamente, para la estructura **`GTree`**.

ATENCIÓN: recordar que no siguen la secuencia dada para tipos recurslvos con recursión directa, porque la recursión en **`GTree`** NO es directa. Estas versiones deben tener en cuenta el esquema de recursión sobre listas correspondiente, como se trató en clase teórica.

b. Definir las siguientes funciones sin utilizar recursión explícita:
   i. `mapGT :: (a -> b) -> GTree a -> GTree b`
   ii. `sumGT :: GTree Int -> Int`
   iii. `sizeGT :: GTree a -> Int`
   iv. `heightGT :: GTree a -> Int`
   v. `preOrderGT :: GTree a -> [a]`
   vi. `postOrderGT :: GTree a -> [a]`
   vii. `mirrorGT :: GTree a -> GTree a`
   viii. `countByGT :: (a -> Bool) -> GTree a -> Int`
   ix. `partitionGT :: (a -> Bool) -> GTree a -> ([a], [a])`
   x. `zipWithGT :: (a->b->c) -> GTree a -> GTree b -> GTree c`
   xi. `caminoMasLargoGT :: GTree a -> [a]`
   xii. `todosLosCaminosGT :: GTree a -> [[a]]`
   xiii. `todosLosNivelesGT :: GTree a -> [[a]]`
   xiv. `caminoHastaGT :: Eq a => a -> GTree a -> [a]`
   xv. `nivelNGT :: GTree a -> Int -> [a]`

Ejercicio 8) Dadas las siguientes definiciones para representar un filesystem:

```
type Name = String
type Content = String
type Path = [Name]
data FileSystem = File Name Content
                | Folder Name [FileSystem]
```

a. Definir **`foldFS`** y **`recFS`**, que expresan la recursión estructural y primitiva, respectivamente, para la estructura **`FileSystem`**.
b. Definir las siguientes funciones sin utilizar recursión explícita:
   i. **`amountOfFiles :: FileSystem -> Int`**, que describe la cantidad de archivos en el filesystem dado.
   ii. **`find :: Name -> FileSystem -> Maybe Content`**, que describe el contenido del archivo con el nombre dado en el filesystem dado.
   iii. **`pathOf :: Name -> FileSystem -> Path`**, que describe la ruta desde la raiz hasta el nombre dado en el filesystem dado.
   Precondición: el nombre existe en el filesystem.
   iv. **`mapContents :: (Content -> Content) -> FileSystem -> FileSystem`**, que describe el filesystem resultante de transformar todos los archivos en el filesystem dado aplicando la función dada.
   v. **`targetedMapContents :: [(Name, Content -> Content)] -> FileSystem -> FileSystem`**, que describe el filesystem resultante de transformar el filesystem dado aplicando la función asociada a cada archivo en la lista dada.