Práctica 5

1. Las listas finitas pueden especificarse como un TAD con los constructores:

- nil: Construye una lista vacía.
- cons: Agrega un elemento a la lista.

y las siguientes operaciones:

- null : Nos dice si la lista es vacía o no.
- head: Devuelve el primer elemento de la lista.
- tail: Devuelve todos los elementos de la lista menos el primero.

a) Dar una especificación algebraica del TAD listas finitas.

b) Dar una especificación tomando como modelo las secuencias.

c) Asumiendo que A es un tipo con igualdad, especificar una función $\mathsf{inL} : \mathsf{List\ A} \to \mathsf{A} \to \mathsf{Bool}$ tal que $\mathsf{inL}\ ls\ x = \mathsf{true}$ si y sólo si x es un elemento de ls.

d) Especificar una función que elimina todas las ocurrencias de un elemento dado.

2. Dado el TAD pilas, con las siguientes operaciones:

- empty: Construye una pila inicialmente vacía.
- push: Agrega un elemento al tope de la pila.
- isEmpty: Devuelve verdadero si su argumento es una pila vacía, falso en caso contrario.
- top: Devuelve el elemento que se encuentra al tope de la pila.
- pop: Saca el elemento que se encuentra al tope de la pila.

Dar una especificación algebraica del TAD pilas y una especificación tomando como modelo las secuencias.

3. Asumiendo que A es un tipo con igualdad, completar la siguiente especificación algebraica del TAD conjunto.

tad Conjunto (A : Set) **where**
  import Bool
  vacío : Conjunto A
  insertar : A → Conjunto A → Conjunto A
  borrar : A → Conjunto A → Conjunto A
  esVacío : Conjunto A → Bool
  unión : Conjunto A → Conjunto A → Conjunto A
  intersección : Conjunto A → Conjunto A → Conjunto A
  resta : Conjunto A → Conjunto A → Conjunto A

$\mathsf{insertar}\ x\ (\mathsf{insertar}\ x\ c) = \mathsf{insertar}\ x\ c$
$\mathsf{insertar}\ x\ (\mathsf{insertar}\ y\ c) = \mathsf{insertar}\ y\ (\mathsf{insertar}\ x\ c)$
...

¿Que pasaría si se agregase una función $\mathsf{choose} : \mathsf{Conjunto\ A} \to \mathsf{A}$, tal que $\mathsf{choose}\ (\mathsf{insertar}\ x\ c) = x$?

4. El TAD *priority queue* es una cola en la cual cada elemento tiene asociado un valor que es su *prioridad* (a dos elementos distintos le corresponden prioridades distintas). Los valores que definen la prioridad de los elementos pertenecen a un conjunto ordenado. Las siguientes son las operaciones soportadas por este TAD:

- vacía: Construye una priority queue vacía.
- poner: Agrega un elemento a una priority queue con una prioridad dada.
- primero: Devuelve el elemento con mayor prioridad de una priority queue.
- sacar: Elimina de una priority queue el elemento con mayor prioridad.
- esVacía: Determina si una priority queue es vacía.
- unión: Une dos priority queues.

Dar una especificación algebraica del TAD priority queue y una especificación tomando como modelo los conjuntos.

5. Agregar a la siguiente definición del TAD árboles balanceados una especificación algebraica para las operaciones size y expose:

$$
\begin{array}{l}
\textbf{tad}\ \mathsf{BalT}\ (\mathsf{A} : \mathsf{Ordered\ Set})\ \textbf{where} \\
\quad \textbf{import}\ \mathsf{Maybe} \\
\quad \mathsf{empty}\ : \mathsf{BalT\ A} \\
\quad \mathsf{join}\ \ \ \ : \mathsf{BalT\ A} \to \mathsf{Maybe\ A} \to \mathsf{BalT\ A} \to \mathsf{BalT\ A} \\
\quad \mathsf{size}\ \ \ \ : \mathsf{BalT\ A} \to \mathsf{N} \\
\quad \mathsf{expose} : \mathsf{BalT\ A} \to \mathsf{Maybe}\ (\mathsf{BalT\ A}, \mathsf{A}, \mathsf{BalT\ A})
\end{array}
$$

- La operación join toma un árbol L, un elemento opcional, y un árbol R. Si L y R son árboles de búsqueda balanceados tales que todos los elementos de L sean menores que todos los elementos de R y el elemento opcional es más grande que los de L y menor que los de R, entonces join crea un nuevo árbol de búsqueda balanceado.
- Las operaciones empty y size son obvias.
- La operación expose toma un árbol T y nos da Nothing si el árbol está vacío, y en otro caso nos devuelve un árbol izquierdo, un elemento raíz, y un árbol derecho de un árbol de búsqueda que contiene todos los elementos de T.

Notar que join no es simplemente un constructor sino que tiene que realizar cierto trabajo para devolver un árbol balanceado. Debido a esto es conveniente especificar expose por casos sobre su resultado.

6. Demostrar que $(\mathsf{uncurry\ zip}) \circ \mathsf{unzip} = \mathsf{id}$, siendo:

$$
\begin{array}{ll}
\mathsf{zip} & :: [a] \to [b] \to [(a,b)] \\
\mathsf{zip}\ [\,]\ ys & = [\,] \\
\mathsf{zip}\ (x : xs)\ [\,] & = [\,] \\
\mathsf{zip}\ (x : xs)\ (y : ys) & = (x,y) : \mathsf{zip}\ xs\ ys \\
\mathsf{unzip} & :: [(a,b,)] \to ([a],[b]) \\
\mathsf{unzip}\ [\,] & = ([\,],[\,]) \\
\mathsf{unzip}\ ((x,y) : ps) & = (x : xs, y : ys) \\
 & \quad \textbf{where}\ (xs, ys) = \mathsf{unzip}\ ps
\end{array}
$$

7.

Demostrar que $\mathsf{sum}\ xs \leqslant \mathsf{length}\ xs * \mathsf{maxl}\ xs$, sabiendo que xs es una lista de números naturales y que maxl y sum se definen

$$
\begin{array}{llll}
\mathsf{maxl}\ [\,] & = 0 & \qquad \mathsf{sum}\ [\,] & = 0 \\
\mathsf{maxl}\ (x : xs) & = x\ `\mathsf{max}`\ \mathsf{maxl}\ xs & \qquad \mathsf{sum}\ (x : xs) & = x + \mathsf{sum}\ xs
\end{array}
$$

8. Dado el siguiente tipo de datos

$$\textbf{data}\ \mathsf{Arbol}\ a = \mathsf{Hoja}\ a \mid \mathsf{Nodo}\ a\ (\mathsf{Arbol}\ a)\ (\mathsf{Arbol}\ a)$$

a) Dar el tipo y definir la función size que calcula la cantidad de elementos que contiene un $(\mathsf{Arbol}\ a)$.

b) Demostrar la validez de la siguiente propiedad: $\forall t \in (\mathsf{Arbol}\ a).\, \exists k \in \mathbb{N}.\, \mathsf{size}\ t = 2\ k + 1$

c) Dar el tipo y definir la función mirror que dado un árbol devuelve su árbol espejo.

d) Demostrar la validez de la siguiente propiedad: $\mathsf{mirror} \circ \mathsf{mirror} = \mathsf{id}$

e) Considerando las siguientes funciones:

$$\begin{array}{ll}
\mathsf{hojas} & :: \mathsf{Arbol}\ a \rightarrow \mathsf{Int} \\
\mathsf{hojas}\ (\mathsf{Hoja}\ x) & = 1 \\
\mathsf{hojas}\ (\mathsf{Nodo}\ x\ t_1\ t_2) & = \mathsf{hojas}\ t_1 + \mathsf{hojas}\ t_2 \\
\mathsf{altura} & :: \mathsf{Arbol}\ a \rightarrow \mathsf{Int} \\
\mathsf{altura}\ (\mathsf{Hoja}\ x) & = 1 \\
\mathsf{altura}\ (\mathsf{Nodo}\ x\ t_1\ t_2) & = 1 + (\mathsf{altura}\ t_1\ \text{`}\mathsf{max}\text{`}\ \mathsf{altura}\ t_2)
\end{array}$$

Demostrar que para todo árbol finito t se cumple que $\mathsf{hojas}\ t < 2^{(\mathsf{altura}\ t)}$

9. Dadas las siguientes definiciones:

$\textbf{data}\ \mathsf{AGTree}\ a = \mathsf{Node}\ a\ [\mathsf{AGTree}\ a]$

$\mathsf{ponerProfs}\ n\ (\mathsf{Node}\ x\ xs) = \mathsf{Node}\ n\ (\mathsf{map}\ (\mathsf{ponerProfs}\ (n+1))\ xs)$

a) Definir una función $\mathsf{alturaAGT}$ que calcule la altura de un AGTree.

b) Definir una función maxAGT que dado un AGTree de enteros devuelva su mayor elemento.

c) Demostrar que $\mathsf{alturaAGT} = \mathsf{maxAGT} \circ \mathsf{ponerProfs}\ 1$

10. Dadas las siguientes definiciones

$\textbf{data}\ \mathsf{Tree}\ a = \mathsf{Leaf}\ a \mid \mathsf{Node}\ a\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)$

$$\begin{array}{ll}
\mathsf{flatten}\ (\mathsf{Leaf}\ x) & = [x] \\
\mathsf{flatten}\ (\mathsf{Node}\ x\ lt\ rt) & = \mathsf{flatten}\ lt \mathbin{+\!\!+} [x] \mathbin{+\!\!+} \mathsf{flatten}\ rt \\
\mathsf{mapTree}\ f\ (\mathsf{Leaf}\ x) & = \mathsf{Leaf}\ (f\ x) \\
\mathsf{mapTree}\ f\ (\mathsf{Node}\ x\ lt\ rt) & = \mathsf{Node}\ (f\ x)\ (\mathsf{mapTree}\ f\ lt)\ (\mathsf{mapTree}\ f\ rt)
\end{array}$$

demostrar que $\mathsf{map}\ f \circ \mathsf{flatten} = \mathsf{flatten} \circ \mathsf{mapTree}\ f$

11. Dada las siguientes definiciones

$$\begin{array}{ll}
\mathsf{join}\ [\,] & = [\,] \\
\mathsf{join}\ (xs : xss) & = xs \mathbin{+\!\!+} \mathsf{join}\ xss
\end{array}$$

$\mathsf{singleton}\ x = [x]$

demostrar

a) $\mathsf{id} = \mathsf{join} \circ \mathsf{map\ singleton}$

b) $\mathsf{join} \circ \mathsf{join} = \mathsf{join} \circ \mathsf{map\ join}$

12. Dadas las funciones $\mathsf{insert} :: \mathsf{Ord}\ a \Rightarrow a \rightarrow \mathsf{Bin}\ a \rightarrow \mathsf{Bin}\ a$, que agrega un elemento a un BST dado, y $\mathsf{inorder} :: \mathsf{Ord}\ a \Rightarrow \mathsf{Bin}\ a \rightarrow [a]$, que realiza un recorrido *inorder* sobre un BST, dadas en clase de teoría, probar las siguientes propiedades sobre las funciones:

a) Si t es un BST, entonces $\mathsf{insert}\ x\ t$ es un BST.

b) Si t es un BST entonces $\mathsf{inorder}\ t$ es una lista ordenada.

13. Dadas las definiciones de funciones que implementan *leftist heaps*, dadas en clase, probar que si l_1 y l_2 son leftist heaps, entonces $\mathsf{merge}\ l_1\ l_2$ es un leftist heap.

14. Dar el principio de inducción estructural para el siguiente tipo de datos.

$\textbf{data}\ \mathsf{F} = \mathsf{Zero} \mid \mathsf{One}\ \mathsf{F} \mid \mathsf{Two}\ (\mathsf{Bool} \rightarrow \mathsf{F})$