Programación Funcional

Trabajo Práctico Nro. 8

Temas: Tipos abstractos de datos y módulos.

Bibliografía relacionada:

- Bird, Richard. Introduction to funtional programming using Haskell. Prentice Hall, 1998 (Second Edition). Cap. 8.
- L.C. Paulson. ML for the working programmer. Cambridge University Press, 1996. Cap. 7.
- Simon Thompson. The craft of Functional Programming. Addison Wesley, 1996. Cap. 12.

1. Sea el TAD de las pilas definido mediante la siguiente signatura:
   Tipos: `Stack a`
   Operaciones:

   - `emptyS :: Stack a`
     que crea una pila vacía.
   - `isEmptyS :: Stack a -> Bool`
     que indica si la pila está vacía.
   - `pushS :: a -> Stack a -> Stack a`
     que devuelve la pila luego de insertar el elemento dado como argumento.
   - `popS :: Stack a -> Stack a`
     que devuelve la pila a la que se le sacó el elemento del tope. La operación es parcial, estando indefinida para la pila vacía.
   - `topS :: Stack a -> a`
     que da el elemento que está al tope de la pila. La operación es parcial, estando indefinida para la pila vacía.
   - `sizeS :: Stack a -> Int`
     que informa el tamaño de la pila.

   Otra forma de especificar el comportamiento de las pilas es mediante las siguientes propiedades:

$$
\begin{array}{lcl}
isEmptyS\ emptyS & = & True \\
isEmptyS\ (pushS\ e\ s) & = & False \\
topS\ (pushS\ e\ s) & = & e \\
popS\ (pushS\ e\ s) & = & s \\
sizeS\ emptyS & = & 0 \\
sizeS\ (pushS\ e\ s) & = & 1\ +\ sizeS\ s
\end{array}
$$

a) Utilizando el TAD de las pilas definir una función que verifique si una cadena de caracteres cumple la condición de balanceo de paréntesis.

b) Hacer lo mismo verificando el balance de corchetes.

c) Finalmente, dar una solución genérica, en la que de acuerdo a los símbolos de apertura y cierre recibidos, indique si una expresión está balanceada.

2. El tipo abstracto de las listas ordenadas se define mediante la siguiente signatura:
Tipos: `SL a`
Operaciones:

   - `buildSL :: (a -> a -> Bool) -> SL a`
     que constuye una nueva lista vacía con un orden indicado.
   - `isEmptySL :: SL a -> Bool`
     que indica si la lista está vacía.
   - `insertSL :: a -> SL a -> SL a`
     que devuelve la lista ordenada a la cual se le inserta el elemento dado como argumento.
   - `headSL :: SL a -> a`
     que da el primer elemento de la lista que precede a todos, de acuerdo al orden.
   - `tailSL :: SL a -> SL a`
     que devuelve la cola de la lista ordenada.

   Utilizando este TAD defina las siguientes funciones.

   Nota: debe usar **sólo** las funciones del TAD.

   - `resort :: (a -> a -> Bool) -> SL a -> SL a`
     que reordena la lista ordenada a partir de otro orden.
   - `sizeSL :: SL a -> Int`
     que devuelve el tamaño de la lista ordenada.
   - `list2SL :: (a -> a -> Bool) -> [a] -> SL a`
     que toma un orden y una lista y devuelve una lista ordenada.
   - `sl2list :: SL a -> [a]`
     que toma una lista ordenada y devuelve una lista del tipo definido en Haskell, que está ordenada.
   - `sortIntSL :: [Int] -> [Int]`
     que dada una lista de enteros, devuelve una en la que los elementos están en orden creciente.

- `sortBySL :: (a -> a -> Bool) -> [a] -> [a]`
  que ordena una lista dada a partir de un orden que recibe como argumento.

3. ¿Qué es un TAD?

4. Implementar los siguientes TADs:

   a) El del ejercicio 1 de la práctica 6 (como TAD).

   b) El del ejercicio 1 de esta práctica. Además, demostrar que las propiedades detalladas de las pilas se satisfacen en la implementación que realizó.

   c) El del ejercicio 2 de esta práctica.

5. ¿Cuales son las ventajas de un lenguaje de programación que soporta módulos?

6. ⊛ El TAD de colecciones de enteros con mínimo se especifica de la siguiente forma

   - `empty :: Col`
     que devuelve una colección de enteros vacía.
   - `insert :: Int -> Col -> Col`
     que inserta un entero en una colección.
   - `lookMin :: Col -> Int`
     que da el entero más chico de la colección. La operación es parcial, estando indefinida cuando la colección no tiene elementos.

   Implementar el tipo de las colecciones con mínimo de tal forma que todas las operaciones utilicen tiempo constante.

Ejercicios complementarios

7. El tipo de datos Lista puede especificarse como TAD mediante la siguiente signatura:
   Tipos: `Lst a`
   Operaciones:

   - `nil :: Lst a`
     que representa a la lista vacía.
   - `size :: Lst a -> Int`
     que devuelve el tamaño de la lista.
   - `at :: Lst a -> Int -> a`
     que devuelve el elemento de la lista en la posición indicada como parámetro. Es parcial si la posición está fuera del rango.
   - `isEmpty :: Lst a -> Bool`
     que indica si la lista es vacía.

- `cons :: a -> Lst a -> Lst a`
  que agrega un elemento a una lista.
- `head' :: Lst a -> a`
  que devuelve el primer elemento de la lista. Es parcial pues está indefinida para la lista vacía.
- `tail' :: Lst a -> Lst a`
  que devuelve la cola de la lista. Es parcial pues está indefinida para la lista vacía.
- `append :: Lst a -> Lst a -> Lst a`
  que devuelve la lista que resulta de la unión de las dos listas que reciben como argumento.
- `map' :: Lst a -> (a->b) -> Lst b`
  que devuelve la lista que resulta de aplicar la función sobre cada elemento de la lista.

Una posible implementación sería con un par, donde la primera componente es su longitud y la segunda es una función, que dado un número, retorna el elemento en esa posición. A modo de ejemplo, se definen la estructura del tipo y las siguientes funciones:

```
module Lst (Lst, nil, size, at,
            isEmpty, cons, head', tail', append) where

data Lst a = L Int (Int -> a)

nil :: Lst a
nil  =  L 0 (const (error "empty"))

size :: Lst a -> Int
size (L n f) =  n

at :: Lst a -> Int -> a
at (L n f) pos = if pos >= 0 && pos < n
                 then f pos
                 else error "Posición fuera de rango"
```

Complete la implementación del TAD.