Recuperatorio de Parcial – Estructuras de Datos – UNQ

Aclaraciones:

- *Esta evaluación es a libro abierto. Se pueden usar todas las funciones y tipos de datos vistos en la práctica y en la teórica, salvo que el enunciado indique lo contrario.*
- *No se olvide de poner nombre, nro. de alumno, nro. de hoja y cantidad total de hojas en cada una de las hojas.*
- *Le recomendamos leer el enunciado en su totalidad y organizar sus ideas antes de comenzar la resolución.*
- *Recuerde que la intención es medir cuánto comprende usted del tema. Por ello, no dude en escribir todo lo que sabe, en explicar lo que se propone antes de escribir código, en probar sus funciones con ejemplos, etc.*

Introducción

Una Random Access List (RAL), cuya traducción sería *lista de acceso aleatorio*, es un tipo de listas que soporta las operaciones lookup y update, que permiten buscar y modificar elementos de posiciones arbitrarias. Implementaremos esta estructura en el lenguaje Haskell.

Implementación

Para representar una RAL utilizaremos un árbol binario. Al árbol lo iremos llenando por niveles, cada nivel de izquierda a derecha, por lo que necesitaremos además una forma de saber en qué *próxima posición* debemos agregar o consumir un elemento. Todo esto queda expresado por la siguiente representación en Haskell:

```
data Tree a   = EmptyT | NodeT (Tree a) a (Tree a)
type Camino   = [Dir]
data Dir      = Izq | Der

data RAL a = MKR (Tree a) Camino
```

La interfaz del tipo abstracto RAL es la siguiente:

- `nil     :: RAL a`
  Devuelve una RAL vacía.
- `isNil   :: RAL a -> Bool`
  Indica si la RAL está vacía.
- `last    :: RAL a -> a`
  Devuelve el último elemento agregado.
- `snoc    :: a -> RAL a -> RAL a`
  Agrega un elemento al final de la RAL.
- `init    :: RAL a -> RAL a`
  Quita el último elemento de la RAL.
- `lookup  :: Int -> RAL a -> a`
  Dada una posición válida obtiene el elemento que se encuentre en esa posición.
- `update  :: Int -> a -> RAL a -> RAL a`
  Dados una posición válida y un elemento, reemplaza el elemento encontrado en esa posición por el nuevo.

Como puede observarse, en esta interfaz se agregan y se borran elementos por el final de la lista y no por el principio.

Ejercicio 1 Implementar las operaciones de la interfaz de RAL, utilizando la representación descrita anteriormente. Para esto, serán útiles las siguientes funciones auxiliares, que pueden considerase como ya implementadas:

- `next :: Camino -> Camino`
  Dado un camino nos devuelve el camino a la posición siguiente en un árbol.
- `prev :: Camino -> Camino`
  Dado un camino nos devuelve el camino a la posición previa en un árbol. Esta operación es parcial si el camino es una la lista vacía.
- `numACamino :: Int -> Camino`
  Dado un índice nos devuelve el camino que conduce hasta el elemento almacenado en esa posición.

Tener en cuenta las siguientes pistas a la hora de implementar las funciones de la interfaz:

- Recordar que el camino que se guarda en estructura es la próxima posición en la que insertaremos un elemento, y siempre conduce a un EmptyT. Si queremos obtener o borrar el último agregado, entonces el camino que conduce el nodo que lo contiene es el camino previo al que tenemos guardado. Este otro camino siempre conduce un árbol no vacío que contiene como raiz al último elemento.
- Actualizar el camino siempre que se borran o agregan elementos (pensar en los invariantes que deben mantenerse).
- Definir subtareas para que la búsqueda, agregado, borrado o actualización de un elemento se simplifiquen. Estas subtareas consisten en recorrer un árbol siguiendo un camino, para realizar, al terminar de recorrer el camino, una determinada tarea. Además, en su implementación, una lista vacía significa que estamos en el nodo o árbol vacío en el que debemos efectuar la operación en particular.

Utilización del tipo abstracto

Ejercicio 2 Como usuario de un RAL se pide implementar las siguientes funciones:

a) `indexOf :: Eq a => a -> RAL a -> Int`
Dado un elemento y un RAL, devuelve la posición de la primera aparición de ese elemento en la RAL.

b) `ralToList :: RAL a -> [a]`
Convierte una RAL en una lista de Haskell, manteniendo el orden de los elementos.

c) `append :: RAL a -> RAL a -> RAL a`
Concatena dos RAL, manteniendo el orden de los elementos. Primero los elementos de la primera lista, y luego los de la segunda.