Reporte Páctica 0

Alejandro Espino Gutiérez
David Ortega Medina

Lenguajes de Programación

1 Funciones

1. `buscar::[Int]->Int->Bool`
   Verifica si un número está en una lista de enteros y devuelve `True` o `False`.
   Nuestro caso base es si la lista esta vacía `[]` y devuelve `False`.
   Nuestro caso recursivo es: si el primer elemento `(y)` es igual a `x`, devuelve `True`, si no sigue buscando en el resto de la lista.
   La sintaxis que usamos es Pattern matching `(y:ys)` para dividir la lista y usamos guardas ‖

2. `sumar_lista::[Int]->Int`
   Calcula la suma de los elementos de una lista de enteros y devuelve un número entero.
   La función principal `sumar_lista` llama a una función auxiliar `sumar_lista_aux` con un acumulador inicializado en `0`.

   (a) `sumar_lista_aux::[Int]->Int->Int`
   La función auxiliar toma una lista de enteros y un acumulador, devolviendo la suma total.
   Nuestro caso base es si la lista está vacía `[]` y devuelve el acumulador actual.
   Nuestro caso recursivo es: si la lista tiene al menos un elemento `(y:ys)`, sumamos `y` al acumulador y seguimos con el resto de la lista.
   La sintaxis que usamos es Pattern matching `(y:ys)` para dividir la lista y aplicamos recursión de cola.

3. Implementa una función recursica de forma ordinario y implementala con recursión de cola

   (a) `maximoListaOrd ::  [Int] -> Int`
   Calcula el máximo de una lista de enteros usando recursión normal.
   Nuestro caso base ocurre cuando la lista está vacía `[]`, la función genera un error indicando que la lista no puede estar vacía.
   Otro caso base es cuando la lista tiene un solo elemento `[x]`, devuelve

`x`.
Nuestro caso recursivo es: si la lista tiene más elementos `(x:xs)`, calcula el máximo entre `x` y el máximo del resto de la lista `maximoListaOrd xs`.
La sintaxis que usamos es Pattern matching `(x:xs)` para dividir la lista y la función `max` para comparar los valores.

(b) `maximoListaCola :: [Int] -> Int`
Calcula el máximo de una lista de enteros usando recursión de cola.
Nuestro caso base ocurre cuando la lista está vacía `[]`, la función genera un error indicando que la lista no puede estar vacía.
En el caso de una lista no vacía `(x:xs)`, la función principal `maximoListaCola` llama a la función auxiliar `maximoListaAux` con el primer elemento como acumulador.

(c) `maximoListaAux :: [Int] -> Int -> Int`
La función auxiliar toma una lista y un acumulador que guarda el máximo encontrado hasta el momento.
Nuestro caso base es cuando la lista está vacía `[]`, devolvemos el valor del acumulador, ya que representa el máximo de todos los elementos procesados.
Nuestro caso recursivo es: si la lista tiene al menos un elemento `(x:xs)`, comparamos `x` con el acumulador y llamamos recursivamente con el resto de la lista, actualizando el acumulador con el máximo entre `x` y su valor actual.

(d) **Rendimiento**
Se ejecutaron ambas versiones con el comando `:s +t` en `ghci` para medir el tiempo y la memoria utilizados.
La función que hicimos en recursión ordinaria se llama 'maximoListaOrd' y al llamarla sobre una lista de 40 elementos '[1, 2, 3, ..., 40]' y usar el comando de estadísticas obtenemos que tardó 0.01 segundos y usó 74,352 bytes.
La función que funciona con recursión de cola se llama 'maximoListaCola' y al llamarla sobre la misma lista de 40 elementos que la recursión ordinaria, obtuvimos que tardó 0.01 segundos y usó 74,400 bytes.
Vemos que existe una diferencia de 48 bytes en favor de la recursión ordinaria. Esto pasa porque la recursión ordinaria guarda los máximos en la pila, mientras que la recursión de cola tiene una pila que crece más lento, debido al efecto que genera el acumulador.

Como lo vimos en el laboratorio la recursión de cola suele ser más eficiente que la recursión ordinaria, sin embargo el compilador de Haskell eficienta el proceso de la recursión ordinaria. Sin embargo,

según lo visto en el laboratorio entre más datos estemos ocupando, la recursión de cola es más eficiente en memoria respecto a la ordinaria, y además evita desbordamientos de pila en listas muy grandes.

4. `filterB::(a -> Bool) -> [a] -> [a]`
   La función `filterB` toma un predicado (una función que devuelve un valor booleano) y una lista, devolviendo una nueva lista con los elementos que cumplen la condición.
   Si la lista de entrada está vacía `[]`, la función devuelve una lista vacía, ya que no hay elementos que evaluar.
   Si la lista tiene al menos un elemento `(x:xs)`, evaluamos el primer elemento `x` con el predicado `p`.
   - Si `p x` es `True`, incluimos `x` en la lista resultante y continuamos con el resto de la lista `xs`.
   - Si `p x` es `False`, simplemente continuamos con el resto de la lista sin incluir `x`.
   La función usa pattern matching `(x:xs)` para dividir la lista en cabeza y cola, y guardas `|` para evaluar la condición del predicado.
   Esto permite una implementación clara y eficiente del filtrado.

5. `mapear:: (a->b) -> [a] -> [b]`
   La función `mapear` toma una función como argumento y la aplica a cada elemento de una lista, devolviendo una nueva lista con los resultados.
   Nuestro caso base es: si la lista de entrada está vacía `[]`, la función devuelve una lista vacía, ya que no hay elementos a los que aplicar la función.
   El caso recursivo es: si la lista tiene al menos un elemento `(x:xs)`, aplicamos la función `f` al primer elemento `x` y agregamos el resultado a la lista resultante. Luego, llamamos recursivamente a `mapear` con el resto de la lista `xs`. Se usa `(x:xs)` para dividir la lista en cabeza y cola.

   (a) `mapear_:: (a->b) -> [a] -> [b]`
   Esta es una implementación alternativa de `mapear` usando list comprehension.
   Aquí, generamos una nueva lista aplicando `f` a cada elemento `x` en `list`.

   La versión recursiva construye la lista elemento por elemento, mientras que la versión con list comprehension es más concisa.

6. `preorder:: Tree a -> [a]`
   La función `preorder` realiza un recorrido en preorden de un árbol binario, devolviendo una lista con los elementos del árbol en el orden adecuado.
   - Nuestro caso base es cuando el árbol es `Empty`, devolvemos una lista vacía `[]`.
   - Nuestro caso recursivo es: si el nodo tiene un valor `root` y dos subárboles `left` y `right`, agregamos primero `root` a la lista, seguido del recorrido en preorden del subárbol izquierdo y luego el subárbol derecho.

Se usa pattern matching `(Node root left right)` para descomponer el árbol y la concatenación `++` para construir la lista resultante.

7. `buscar_tree:: Eq a => Tree a -> a -> Bool`
   Verifica si un elemento está presente en un árbol binario.
   - Nuestro caso base es cuando el árbol es `Empty`, si no devolvemos `False`, ya que no hay elementos en el árbol.
   - Nuestro caso recursivo es: si el nodo actual `root` es igual a `e`, devolvemos `True`. Si no, seguimos buscando en los subárboles izquierdo y derecho.
   Se usa pattern matching `(Node root left right)` para dividir el árbol y `||` para buscar en ambos subárboles.

8. `hojas :: Tree a -> Int`
   La función `hojas` cuenta la cantidad de hojas en un árbol binario.
   - Nuestro caso base es cuando el árbol es `Empty`, devolvemos `0`, ya que no hay hojas.
   - Si el nodo es una hoja, o sea un `Node` sin subárboles (`left` y `right` son `Empty`), devolvemos `1`, este nodo es una hoja.
   - Nuestro caso recursivo es: si el nodo tiene subárboles `left` y `right`, sumamos la cantidad de hojas en ambos subárboles, llamando recursivamente a `hojas left` y `hojas right`.
   Se usa pattern matching `(Node root left right)` para descomponer el árbol y la recursión para contar las hojas.

9. `vecinos:: Graph -> Vertex -> [Vertex]`
   La función `vecinos` devuelve la lista de vértices adyacentes a un vértice dado en una gráfica representado como una lista de adyacencia.
   - Nuestro caso base es cuando la lista de adyacencia está vacía `[]`, devolvemos una lista vacía.
   - Nuestro caso recursivo es: si el vértice actual `v` coincide con el vértice buscado `x`, devolvemos su lista de vecinos `ns`. Si no, seguimos buscando en el resto de la lista de adyacencia.
   Se usa pattern matching `((v, ns):xs)` para recorrer la lista de adyacencia y la estructura condicional para verificar si el vértice coincide con el buscado.

10. `dfs:: Graph -> Vertex -> [Vertex] -> [Vertex]`
    La función `dfs` realiza un recorrido en profundidad (*Depth First Search, DFS*) en una gráfica.
    - Nuestro caso base es cuando el vértice `v` ya está en la lista de visitados `visited`, en cuyo caso devolvemos la misma lista sin cambios.
    - Nuestro caso recursivo es: si el vértice `v` no ha sido visitado, lo agregamos a la lista y realizamos una llamada recursiva para cada uno de sus

vecinos.
Se usa la función `foldl` para aplicar recursión acumulativa sobre la lista de vecinos, asegurando que el recorrido se propague a través de todos los vértices conectados.

11. `isConnected :: Graph -> Bool`
    La función `isConnected` verifica si una gráfica es conexo.
    - Nuestro caso base es cuando el gráfica es `[]`, devolvemos `True`, ya que una gráfica vacía se considera conexa.
    - En el caso general, extraemos la lista de vértices con `map fst graph` y usamos `dfs` para visitar todos los vértices accesibles desde el primero de la lista. Si el número de vértices visitados es igual al número total de vértices, la gráfica es conexa.

12. `isTree :: Graph -> Bool`
    La función `isTree` determina si una gráfica es un árbol. Un árbol es una gráfica conexa sin ciclos.
    - Nuestro caso base es cuando el gráfica es `[]`, en cuyo caso devolvemos `True`, ya que una gráfica vacío se considera un árbol.
    - En el caso general, verificamos si la gráfica es conexa (`isConnected graph`) y si no contiene ciclos (`sinCiclos graph`). Si ambas condiciones se cumplen, devolvemos `True`, de lo contrario, `False`.

13. `sinCiclos :: Graph -> Bool`
    La función `sinCiclos` verifica si una gráfica no contiene ciclos.
    - Nuestro caso base es cuando el gráfica es `[]`, en cuyo caso devolvemos `True`.
    - En el caso general, usamos la función auxiliar `tieneCiclo` para cada vértice. Si `tieneCiclo` devuelve `True` para algún vértice, entonces hay un ciclo en el gráfica.

14. `tieneCiclo :: Graph -> Vertex -> [Vertex] -> [Vertex] -> Bool`
    La función `tieneCiclo` detecta si existe un ciclo en una gráfica a partir de un vértice dado.
    - Si el vértice actual `v` ya está en el camino de la exploración, hemos encontrado un ciclo, por lo que devolvemos `True`.
    - Si el vértice ya fue visitado en una búsqueda anterior, devolvemos `False`, ya que no puede formar un ciclo nuevo.
    - En el caso general, agregamos el vértice a la lista de visitados y seguimos explorando sus vecinos recursivamente.

15. `leafSum :: Tree Int -> Int`
    La función `leafSum` calcula la suma de los valores almacenados en las ho-

jas de un árbol binario.
- Nuestro caso base es cuando el árbol es `Empty`, devolvemos `0`, ya que no hay hojas que sumar.
- Si el nodo es una hoja, un `Node` sin subárboles (`left` y `right` son `Empty`), devolvemos el valor almacenado en la raíz `root`, ya que representa el único valor de esa hoja.
- Nuestro caso recursivo es: si el nodo tiene subárboles `left` y `right`, sumamos los valores de las hojas en ambos subárboles, llamando recursivamente a `leafSum left` y `leafSum right`.
Se usa pattern matching `(Node root left right)` para recorrer el árbol de manera recursiva y calcular la suma de sus hojas.

2 Comentarios generales

En esta práctica trabajamos con recursión normal y recursión de cola en Haskell, implementando varias funciones como búsqueda en listas, suma de elementos, cálculo del máximo y recorridos en árboles y grafos.
Nos costó un poco entender cómo funciona el 'dfs' en gráficas, ya que mantener la lista de visitados y asegurarnos de no repetir nodos.
En general, la práctica nos ayudó a comprender mejor la diferencia entre recursión normal y de cola. También reforzamos nuestra comprensión sobre estructuras de datos como listas, árboles y grafos, aplicando distintas estrategias para recorrer y manipular estos elementos de manera eficiente en Haskell.