Práctico 8 de Programación Funcional

UdelaR/FCien/CMat

xx al yy de noviembre de 2016

0.1 Árboles binarios con datos en las hojas

Se considera el tipo de los árboles binarios

data `Btree a = Leaf a | Fork (Btree a) (Btree a)`

y las funciones `size, height, nodes, depths, flatten, mkBtree, maxBtree` vistas en el teórico[1].

1. Se define $f : \mathcal{H} \to \mathcal{H}$ tal que:

   - f(`b`) =`Integer` para todo tipo `b`
   - f(`h`) =`id::Integer -> Integer` para toda `h::b->c`.

   (a) Probar que f es un functor.

   (b) Definir `mapBtree::(a->b) -> Btree a -> Btree b`.

   (c) Probar que `mapBtree` es un (endo)functor de la categoría $\mathcal{H}$ de las funciones Haskell-computables.

   (d) Probar que `size : mapBtree => f` es una transformación natural del functor `mapBtree` en el functor `f`.

2. Probar que `size = length.flatten`

3. ¿Cuántos árboles binarios `xt::Btree Int` con todas sus hojas iguales a `0` y tales que `size xt = 5` hay? Definir una función que, para cada `n::Int`, calcule la cantidad de árboles `xt::Btree Int` tales que `size xt = n` y todas sus hojas son iguales a `0`.

4. Probar que `flatten.mkBtree` es la identidad en listas finitas. ¿Es cierta la recíproca? Justificar.

5. (a) Probar que para todo árbol binario finito `xt::Btree a` se tiene la acotación

$$\ulcorner \log_2(\texttt{size xt}) \urcorner \leq \texttt{height xt} < \texttt{size xt}$$

[1] ver: Introduction to Functional Programming using Haskell. R. Bird

(b) Probar que para todo árbol finito `xt::Btree a` se tiene la acotación

$$\texttt{height xt} \leq \texttt{nodes xt} \leq \sum_{i=0}^{i=\texttt{height xt}} 2^i$$

(c) Concluir que $\ulcorner \log_2(\texttt{nodes xt}) \urcorner \leq \texttt{height xt + 1} \leq \texttt{nodes xt + 1}$.

6. Definir `subtrees :: Btree a -> [Btree a]` que, dado un árbol `xt`, calcule la lista de todos los subárboles de `xt`. Enunciar y probar un resultado que vincule la cantidad de subárboles de un árbol `xt` con su tamaño (i.e.: `length.subtrees xt` con `size xt`).

7. Probar que `(n+).height = maxBtree.down n`. Usarlo para concluir que `height = maxBtree.depths`.

8. (a) Sea `xt::Btree a`. Definir una función

   ```
   levelSize :: Int -> Btree a -> Int
   ```

   que, dado un entero n, calcule la cantidad de datos presentes en el nivel n de `xt`.

   (b) Diremos que el n-ésimo nivel de `xt::Btree a` está *completo* si y sólo si `levelSize n xt` es igual a 2^n. Se dice que `xt` es *balanceado* cuando todo nivel $n <$`height xt` está completo.

   Probar que si `Fork xt yt` es balanceado, entonces `xt` e `yt` son balanceados. ¿Es cierta la recíproca? Justificar.

   (c) Probar que para todo `xt::Btree a` finito de altura h, `xt` es balanceado si y sólo si su nivel $h-1$ está completo. Concluir que `xt` es balanceado si y sólo si

   $$2^{\texttt{(height xt)-1}} < \texttt{size xt} \leq 2^{\texttt{height xt}}$$

   Definir `balancedBtree :: Btree a -> Bool` que en árboles finitos devuelva `True` o `False` según si el árbol es balanceado o no.

   (d) Se define $\mathcal{C}$ como la menor subclase de `BTree a` que satisface:

   - $\forall$`x::a (Leaf x)` $\in \mathcal{C}$.
   - Si `xt, yt` $\in \mathcal{C}$ y `|size yt - size xt|`≤ 1, entonces `(Fork xt yt)` $\in \mathcal{C}$.

   Probar que todos los árboles de $\mathcal{C}$ son balanceados. ¿Es cierta la recíproca? Justificar.

0.2 Árboles binarios de búsqueda (ABB)

Se considera el tipo

data`(Ord a)=>Stree a = Null | Fork (Stree a) a (Stree a)`

y las funciones `size, height, nodes, flatten, member, insert, delete, mkStree` de `Stree a` vistas en el teórico[2].

[2] ver: Introduction to Functional Programming using Haskell. R. Bird

1. Enunciar el principio de inducción estructural para el tipo `Stree a`, explicitando condiciones suficientes para árboles parciales y para árboles finitos.

2. Probar que para todo árbol finito `xt::Stree a` se tiene la acotación

$$\texttt{height xt} \leq \texttt{size xt} < 2^{\texttt{(height xt)}}$$

   probar que esta acotación equivale a

$$\ulcorner \log_2 \texttt{((size xt)+1)} \urcorner \leq \texttt{height xt} < \texttt{(size xt)+1}$$

3. Enunciar y demostrar una acotación similar a la de 2. que relacione `height xt` y `nodes xt` para todo `xt::Stree a` finito.

4. Definir las funciones `mapStree` y `foldStree` para el tipo `Stree a`, declarando sus tipos.

5. Definir usando `foldStree` las funciones

```
headSTree  ::  Stree a -> a
tailSTree  ::  Stree a -> Stree a
```

   tales que:

```
        headSTree  =  head.flatten
flatten.tailSTree  =  tail.flatten
```

   en árboles finitos. ¿Qué sucede con árboles parciales e infinitos?

6. Se consideran las siguientes definiciones

```
           join1, join2  ::  Stree a->Stree a->Stree a
          join1 Null yt   =   yt
join1 (Fork ut x vt) yt   =   Fork ut x (join1 vt yt)

join2 xt yt  |  empty yt   =  xt
             |  otherwise  =

                          Fork xt (headTree yt) (tailTree yt)
```

   Acotar `height (join1 xt yt)` y `height (join2 xt yt)` en función de `height xt` y `height yt`.

7. Probar que `sort::(Ord a)=>[a]->[a]` definida como

```
sort=flatten.mkStree
```

   calculada en una lista finita de `[a]`, ordena en sentido creciente la lista que recibe como argumento.

   Eliminar la etapa intermedia (el árbol) para definir directamente una función `qsort::(Ord a)=>[a]-[a]` que ordene las listas finitas de `[a]`[3].

[3]Este algoritmo se denomina *quicksort* e invierte las listas en tiempo a lo sumo $o(n^2)$, siendo n el largo de la lista. Sin embargo, en promedio, invierte las listas en tiempo $o(n \log_2(n))$