Entrega Práctica 4 EDyA II

Farizano, Juan Ignacio

Ejercicio 4:

tad PriorityQ (a : Set) **where**
 import Bool
 vacia : PriorityQ a
 poner : a $\to$ N $\to$ PriorityQ a $\to$ PriorityQ a
 primero : PriorityQ a $\to$ a
 sacar : PriorityQ a $\to$ PriorityQ a
 esVacia : PriorityQ a $\to$ Bool
 union : PriorityQ a $\to$ PriorityQ a $\to$ PriorityQ a

Especificación algebraica

```
poner x p (poner y p c) = poner y p c


primero (poner x p vacia) = x
primero (poner x p (poner y q c)) = if x > y then primero (poner x p c)
                                              else primero (poner y q c)


sacar (poner x p vacia) = vacia
sacar (poner x p (poner y q c)) = if p > q then poner y q (sacar (poner x p c))
                                            else poner x p (sacar (poner y q c))


esVacia vacia = True
esVacia (poner x p q) = False


union c vacia = c
union c (poner x p q) = union (poner x p c) q
```

Especificación tomando como modelo los conjuntos

vacia = {}

poner x p $\{(x_1, p_1), ..., (x_n, p_n)\} = \{(x_1, p_1), ..., (x_n, p_n)\}$ si $p \in \{p_1, ..., p_n\}$

poner x p $\{(x_1, p_1), ..., (x_n, p_n)\} = \{(x_1, p_1), ..., (x_n, p_n), (x, p)\}$ si $p \notin \{p_1, ..., p_n\}$

primero $\{(x_1, p_1), ..., (x_n, p_n)\} = (x_i, p_i)$ tal que $p_i = max(p_1, ..., p_n)$

sacar A = A - $\{(x_i, p_i) \in A/p_i = max(p_1, ..., p_n)\}$

esVacia $\{(x_1, p_1), ..., (x_n, p_n)\} = True$ si $n = 0$

esVacia $\{(x_1, p_1), ..., (x_n, p_n)\} = False$ si $n > 0$

union $\{(x_1, p_1), ..., (x_n, p_n)\}$ $\{(y_1, q_1), ..., (y_m, q_m)\} = \{(x_1, p_1), ..., (x_n, p_n)\} \cup \{(y_i, q_i)/q_i \notin \{p_1, ..., p_n\}\}$

Ejercicio 9:

Dado el tipo de datos **data** AGTree a = Node a [AGTRee a], defino su principio de inducción estructural:
Dada una propiedad P sobre $AGTree$, para probar $\forall t :: AGTree.\ P(t)$:

- Probamos $P(Node\ a\ [])$
- Probamos que si $P(x_i)\ \forall i = 1, ..., n$ entonces $P(Node\ a\ [x_1, ..., x_n])$

Ejercicio 13:

Definimos el siguiente tipo de datos
type Rank = Int
data Heap a = E | N Rank a (Heap a) (Heap a)

El **rango** de un heap es la longitud de la espina derecha (el camino hacia la derecha hasta un nodo vacío.)
Un leftist heap es una variante de heap cuya invariante es que el rango de cualquier hijo izquierdo es mayor o igual que el de su hermano de la derecha. Dado este tipo de datos, definimos las siguientes funciones:

```
merge :: Ord a => Heap a -> Heap a -> Heap a
merge h1 E = h1
merge E h2 = h2
merge h1@(N _ x a1 b1) h2@(N _ y a2 b2) = if x <= y then makeH x a1 (merge b1 h2)
                                                     else makeH y a2 (merge h1 b2)
```

```
rank :: Heap a -> Rank
rank E = 0
rank (N r _ _ _) = r


makeH x a b = if rank a >= rank b then N (rank b + 1) x a b
                                  else N (rank a + 1) x b a
```

Proposición: Probar que si l_1 y l_2 son leftist heaps, entonces merge l_1 l_2 es un leftist heap.

Para probar utilizaremos inducción estructural, primero definimos el principio de inducción estructural de este tipo de datos:

Dada una propiedad P sobre $Heap$, para probar $\forall\ h :: Heap.\ P(h)$:

- Probamos $P(E)$
- Probamos que si $P(h_1)$ y $P(h_2)$ entonces $P(N\ Rank\ a\ h_1\ h_2)$

Demostración por inducción estructural sobre l_1:

Sea l_1 un leftist heap, luego $P(l_1)$: merge l_1 l_2 es un leftist heap $\forall\ l_2$ leftist heap.

• Caso base: $l_1 \ = \ E$

Sea l_2 un leftist heap.

merge l_1 l_2 $\underset{l_1=\text{E}}{=}$ merge E l_2 $\underset{merge.2}{=}$ l_2

• Caso inductivo: $l_1 = N\ r_1\ x\ a_1\ b_1$

Sea l_2 un leftist heap.

Hipótesis inductivas:

- H1: merge a_1 l_2 es un leftist heap $\forall\ l_2$ leftist heap.
- H2: merge b_1 l_2 es un leftist heap $\forall\ l_2$ leftist heap.

Utilizamos inducción estructural sobre l_2:

• Si $l_2 = $ E:

merge l_1 l_2 $\underset{l_2=\text{E}}{=}$ merge l_1 E $\underset{merge.1}{=}$ l_1

• Si $l_2 = N\ r_2\ y\ a_2\ b_2$ tenemos dos posibilidades: $x \leq y$ ó $x > y$:

Para el caso inductivo de l_2 tenemos nuevas **hipótesis inductivas:**

- H3: merge l_1 a_2 es un leftist heap $\forall\ l_1$ leftist heap.
- H4: merge l_1 b_2 es un leftist heap $\forall\ l_1$ leftist heap.

Vemos primero si $x \leq y$:

merge l_1 l_2 $\underset{merge.3 \text{ y } x \leq y}{=}$ makeH x a_1 (merge b_1 l_2). Por H2 merge b_1 l_2 es un leftist heap, y por **Lema 1** $\forall$ l_1 l_2 leftist heap $\forall x$. makeH x l_1 l_2 es un leftist heap, resultando ser makeH x a_1 (merge b_1 l_2) un leftist heap.

Por último, si $x > y$:

merge l_1 l_2 $\underset{merge.3 \text{ y } x > y}{=}$ makeH y a_2 (merge l_1 b_2). Por H4 merge l_1 b_2 es un leftist heap, y por **Lema 1** $\forall$ l_1 l_2 leftist heap $\forall x$. makeH x l_1 l_2 es un leftist heap, resultando ser makeH x a_2 (merge l_1 b_2) un leftist heap.

Por lo tanto, merge l_1 l_2 es un leftist heap.

Solo queda probar el lema auxiliar:

Lema 1: $\forall$ l_1 l_2 leftist heap $\forall x$. makeH x l_1 l_2 es un leftist heap.

Demostración:

Sean l_1 y l_2 leftist heaps, hay dos casos posbiles: rank $l_1 \geq$ rank l_2 ó rank $l_1 <$ rank l_2.

- Si rank $l_1 \geq$ rank l_2:

makeH x l_1 l_2 $= N\ (rank\ l_2 + 1)\ x\ l_1\ l_2$. Ya que rank $l_1 \geq l_2$, se cumple la invariante y makeH x l_1 l_2 es un leftist heap.

- Si rank $l_1 <$ rank l_2:

makeH x l_1 l_2 $= N\ (rank\ l_1 + 1)\ x\ l_2\ l_1$. Ya que rank $l_1 < l_2$, se cumple la invariante y makeH x l_1 l_2 es un leftist heap.

Por lo tanto, el **Lema 1** queda demostrado.