1. Las listas finitas pueden especificarse como un TAD con los constructores:

   - `Nil`: Construye una lista vacía.
   - `Cons`: Agrega un elemento a la lista.

   y las siguientes operaciones:

   - `null`: Nos dice si la lista es vacía o no.
   - `head`: Devuelve el primer elemento de la lista.
   - `tail`: Devuelve todos los elementos de la lista menos el primero.

   a) Dar una especificación algebraica del TAD listas finitas.

   b) Dar una especificación tomando como modelo las secuencias.

   c) Asumiendo que a es un tipo con igualdad, especificar una función `inL :: List a -> a -> Bool` tal que `inL xs x` ≡ `True` si y solo si `x` es un elemento de `xs`.

   d) Especificar una función que elimina todas las ocurrencias de un elemento dado.

Soluciones

a)

```
TAD List (a :: Set) where
-----------------------------
import Bool
Nil :: List a
Cons :: a -> List a -> List a
null :: List a -> Bool
head :: List a -> a
tail :: List a -> List a
-----------------------------
null Nil           ≡ True
null (Cons x xs)   ≡ False
head (Cons x xs)   ≡ x
tail (Cons x xs)   ≡ xs
```

b)

`Nil`	$\equiv \langle\rangle$
`Cons` $(x\ \langle x_1, \ldots, x_n\rangle)$	$\equiv \langle x, x_1, \ldots, x_n\rangle$
`null` $\langle x_1, \ldots, x_n\rangle$	$\equiv$ `True` (si $n = 0$)
	$\equiv$ `False` (si $n > 0$)
`head` $\langle x_1, \ldots, x_n\rangle$	$\equiv x_1$
`tail` $\langle x_1, \ldots, x_n\rangle$	$\equiv \langle x_2, \ldots, x_n\rangle$

c)

```
inL Nil x             ≡ False
inL (Cons x xs) y     ≡ if x == y then True else inL xs y
```

d)

```
delete Nil x            ≡ Nil
delete (Cons x xs) y    ≡ if x == y then delete xs y
                                    else Cons x (delete xs y)
```

2. Dado el TAD pilas, con las siguientes operaciones:

   - `Empty`: Construye una pila inicialmente vacía.
   - `Push`: Agrega un elemento al tope de la pila.
   - `isEmpty`: Devuelve verdadero si su argumento es una pila vacía, falso en caso contrario.
   - `top`: Devuelve el elemento que se encuentra al tope de la pila.
   - `pop`: Saca el elemento que se encuentra al tope de la pila.

   Dar una especificación algebraica del TAD pilas y una especificación tomando como modelo las secuencias.

Soluciones

- TAD `Stack (a :: Set)` where

```
import Bool
Empty :: Stack a
Push :: a -> Stack a -> Stack a
isEmpty :: Stack a -> Bool
top :: Stack a -> a
pop :: Stack a -> Stack a
```

```
isEmpty Empty          ≡ True
isEmpty (Push x xs)    ≡ False
top (Push x xs)        ≡ x
pop (Push x xs)        ≡ xs
```

`Empty`	$\equiv \langle\rangle$
`Push` $(x\ \langle x_1, \ldots, x_n\rangle)$	$\equiv \langle x, x_1, \ldots, x_n\rangle$
`isEmpty` $\langle x_1, \ldots, x_n\rangle$	$\equiv$ `False` (si $n > 0$)
	$\equiv$ `True` (si $n = 0$)
`top` $\langle x_1, \ldots, x_n\rangle$	$\equiv x_1$
`pop` $\langle x_1, \ldots, x_n\rangle$	$\equiv \langle x_2, \ldots, x_n\rangle$

3. Asumiendo que a es un tipo con igualdad, completar la siguiente especificación algebraica del TAD conjunto.

   TAD `Conjunto (a :: Set)` where

   ```
   import Bool
   Vacio :: Conjunto a
   Insertar :: a -> Conjunto a -> Conjunto a
   borrar :: a -> Conjunto a -> Conjunto a
   esVacio :: Conjunto a -> Bool
   union :: Conjunto a -> Conjunto a -> Conjunto a
   interseccion :: Conjunto a -> Conjunto a -> Conjunto a
   resta :: Conjunto a -> Conjunto a -> Conjunto a
   ```

   ¿Que pasaría si se agregase una función `choose :: Conjunto a -> a`, tal que `choose (insertar x c)` $\equiv$ `x`?

Solución

```
insertar x (insertar y c)          ≡ insertar y (insertar x c)
insertar x (insertar x c)          ≡ insertar x c
borrar x Vacio                     ≡ Vacio
borrar x (insertar y c)            ≡ if x == y
                                     then borrar x c
                                     else insertar y (borrar x c)
esVacio Vacio                      ≡ True
esVacio (insertar x xs)            ≡ False
union c Vacio                      ≡ c
union c (insertar x c')            ≡ union (insertar x c) c'
in x Vacio                         ≡ False
in x (insertar y c)                ≡ if x == y then True
                                               else in x c
interseccion c Vacio               ≡ Vacio
interseccion c (insertar x c')     ≡ if x in c
                                     then insertar x (interseccion c c')
                                     else interseccion c c'
resta c Vacio                      ≡ c
resta c (insertar x c')            ≡ resta (borrar x c) c'
inL (Cons x xs) y                  ≡ if x == y then True else inL xs y
```

4. El TAD *priority queue* es una cola en la cual cada elemento tiene asociado un valor que es su prioridad (a dos elementos distintos le corresponden prioridades distintas). Los valores que definen la prioridad de los elementos pertenecen a un conjunto ordenado. Las siguientes son las operaciones soportadas por este TAD:

   - `Vacia`: Construye una priority queue vacía.
   - `Poner`: Agrega un elemento a una priority queue con una prioridad dada.
   - `primero`: Devuelve el elemento con mayor prioridad de una priority queue.
   - `sacar`: Elimina de una priority queue el elemento con mayor prioridad.
   - `esVacia`: Determina si una priority queue es vacía.
   - `union`: Une dos priority queues.

Dar una especificación algebraica del TAD priority queue y una especificación tomando como modelo los conjuntos.

Soluciones

- ```
  TAD PQueue (a :: Set) where
  ---------------------------------------------------------------
  import Bool, Nat
  Vacia :: PQueue a
  Poner :: a -> Nat -> PQueue a -> PQueue a
  primero :: PQueue a -> a
  sacar :: PQueue a -> PQueue a
  esVacia :: PQueue a -> Bool
  union :: PQueue a -> PQueue a
  ---------------------------------------------------------------
  poner x p (poner y p c)            ≡ poner x p c
  poner x p (poner y q c)            ≡ poner y q (poner x p c)
  primero (Poner x p Vacia)          ≡ x
  primero (Poner x p (Poner y q c))  ≡ if p > q
                                         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 c)              ≡ False
  union Vacia q                      ≡ q
  union (Poner x p c) q              ≡ Poner x p (union c q)
  ```

`Vacia`	$\equiv \{\}$
`Poner` $(x\ p\ C)$	$\equiv \{(x,p)\} \cup \{(a,b) \in C / b \neq p\}$
`primero` $C = \{(x_1,p_1), \ldots, (x_n,p_n)\}$	$\equiv (x_i,p_i)$ donde $p_i = \text{máx}\{p_1, \ldots, p_n\}$
`sacar` $C = \{(x_1,p_1), \ldots, (x_n,p_n)\}$	$\equiv \{(x_i,p_i) \in C / p_i \neq \text{máx}\{p_1, \ldots, p_n\}\}$
`esVacia` $\{(x_1,p_1), \ldots, (x_n,p_n)\}$	$\equiv$ `False` (si $n > 0$)
	$\equiv$ `True` (si $n = 0$)
`union` $A = \{(a_1,b_1), \ldots, (a_n,b_n)\}\ B$	$\equiv A \cup \{(x,y) \in B / y \notin \{b_1, \ldots, b_n\}\}$
  ```

5. Agregar a la siguiente definición del TAD árboles balanceados, una especificación para las operaciones `size` y `expose`:

```
TAD BalT (a :: Ordered Set) where
import Maybe, Nat, Tupla2
Empty :: BalT a
join :: BalT a -> Maybe a -> BalT a -> BalT a
size :: BalT a -> Nat
expose :: BalT a -> Maybe (BalT a, a, BalT a)
```

- La operación `join` toma un árbol `l`, un elemento opcional y un árbol `r`. Si `l` y `r` son árboles de búsqueda balanceados tales que todos los elementos de `l` sean menores que todos los elementos de `r` y el elemento opcional es más grande que los de `l` y menor que los de `r`, entonces `join` crea un nuevo árbol de búsqueda balanceado.
- Las operaciones `Empty` y `size` son obvias.
- La operación `expose` toma un árbol `t` y nos da `Nothing` si el árbol está vacío, y en otro caso nos devuelve un árbol izquierdo, un elemento raíz y un árbol derecho de un árbol de búsqueda que contiene todos los elementos de `t`.

Notar que `join` no es simplemente un constructor sino que tiene que realizar cierto trabajo para devolver un árbol balanceado. Debido a esto es conveniente especificar `expose` por casos sobre su resultado.

Solución

```
size Empty                  ≡ 0
size (join l Nothing r)     ≡ size l + size r
size (join l (Just x) r)    ≡ size l + size r + 1
expose t                    ≡ case (expose t) of
                              Nothing -> t = Empty
                              Just (l, x, r) -> t = join l (Just x) r
```

6. Demostrar que `(uncurry zip) . unzip` $\equiv$ `id`, siendo:

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys          = []
zip (x:xs) []      = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

unzip :: [(a,b)] -> ([a], [b])
unzip []          = ([], [])
unzip ((x,y):ps) = (x:xs, y:ys)
                    where (xs,ys) = unzip ps

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f p = f (fst p) (snd p)

(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f = \p -> f (fst p) (snd p)
```

Solución

- Caso base `[]`:

  `((uncurry zip) . unzip) []`

  $\equiv \langle def.\circ \rangle$

  `(uncurry zip) (unzip [])`

  $\equiv \langle def.unzip.1 \rangle$

  `(uncurry zip) ([], [])`

  $\equiv \langle def.uncurry \rangle$

  `zip (fst ([], [])) (snd ([], []))`

  $\equiv \langle def.fst; def.snd \rangle$

  `zip [] []`

  $\equiv \langle def.zip.1 \rangle$

  `[]`

  $\equiv \langle def.id \rangle$

  `id []`

- Caso inductivo `(x,y):ps`:

  `((uncurry zip) . unzip) ((x,y):ps)`

  $\equiv \langle def.\circ \rangle$

  `(uncurry zip) (unzip ((x,y):ps))`

  $\equiv \langle def.unzip.2 \rangle$

  `(uncurry zip) (x:xs, y:ys) where (xs, ys) = unzip ps`

  $\equiv \langle def.uncurry; def.fst; def.snd \rangle$

  `zip (x:xs) (y:ys) where (xs, ys) = unzip ps`

  $\equiv \langle def.zip.3; def.where \rangle$

  `(x,y) : zip (unzip ps)`

  $\equiv \langle def.\circ; H.I.; def.id \rangle$

  `(x,y):ps`

  $\equiv \langle def.id \rangle$

  `id ((x,y):ps)`

7. Demostrar que `sum xs` $\leq$ `length xs` $*$ `maxl xs`, sabiendo que `xs` es una lista de números naturales y que `maxl` y `sum` se definen como:

```
maxl :: Ord a => [a] -> a
maxl []     = 0
maxl (x:xs) = x `max` maxl xs

sum :: Num a => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs

length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

Solución

- Caso base `[]`:

```
sum []
```
$\equiv \langle def.sum.1 \rangle$
```
0
```
$\equiv \langle def.* \rangle$
```
0 * 0
```
$\leq \langle prop. \leq \rangle$
```
0 * 0
```
$\equiv \langle def.length.1; def.maxl.1 \rangle$
```
length [] * maxl []
```

- Caso inductivo `x:xs`:

```
sum (x:xs)
```
$\equiv \langle def.sum.2 \rangle$
```
x + sum xs
```
$\leq \langle H.I. \rangle$
```
x + (length xs * maxl xs)
```
$\leq$
```
x + (x `max` maxl xs) * (length xs)
```
$\leq$
```
(x `max` maxl xs) + (x `max` maxl xs) * (length xs)
```
$\equiv \langle prop.* \rangle$
```
(1 + length xs) * (x `max` maxl xs)
```
$\equiv \langle def.length.2 + def.maxl.2 \rangle$
```
length (x:xs) * maxl (x:xs)
```

8. Dado el siguiente tipo de datos: `data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)`

   a) Dar el tipo y definir la función `size` que calcula la cantidad de elementos que contiene un `Arbol a`.

   b) Demostrar la validez de la siguiente propiedad: $\forall t \in$(`Arbol a`) $\exists k \in \mathbb{N}/$`size` $t = 2k + 1$.

   c) Dar el tipo y definir la función `mirror` que dado un árbol devuelve su árbol espejo.

   d) Demostrar la validez de la siguiente propiedad: `(mirror . mirror)` $\equiv$ `id`

   e) Considerando las siguientes funciones:

```
hojas :: Arbol a -> Int
hojas (Hoja x)       = 1
hojas (Nodo x t1 t2) = hojas t1 + hojas t2
altura :: Arbol a -> Int
altura (Hoja x)       = 1
altura (Nodo x t1 t2) = 1 + (altura t1 `max` altura t2)
```

   Demostrar que para todo árbol finito `t` se cumple que `hojas t` $<$ `2 ^ (altura t)`

Soluciones

a)
```
size :: Arbol a -> Int
size (Hoja _)     = 1
size (Nodo _ l r) = 1 + size l + size r
```

b)

- Caso base `Hoja x`:

  `size (Hoja x)`

  $\equiv \langle def.size.1 \rangle$

  `1`

  $\equiv \langle k = 0 \rangle$

  `2k + 1`

- Caso inductivo `Nodo x l r`:

```
size (Nodo _ l r)
```
$\equiv \langle def.size.2 \rangle$
```
1 + size l + size r
```
$\equiv \langle H.I. \rangle$
```
1 + (2k'+1) + (2k''+1)
```
$\equiv \langle k = k' + k'' + 1 \rangle$
```
2k+1
```

c)
```
mirror :: Arbol a -> Arbol a
mirror (Hoja x)     = Hoja x
mirror (Nodo x l r) = Nodo x (mirror r) (mirror l)
```

d)

- Caso base `Hoja x`:

```
(mirror . mirror) (Hoja x)
```
$\equiv \langle def.\circ \rangle$
```
mirror (mirror (Hoja x))
```
$\equiv \langle def.mirror.1; def.mirror.1 \rangle$
```
Hoja x
```
$\equiv \langle def.id \rangle$
```
id (Hoja x)
```

- Caso inductivo `Nodo x l r`:

```
(mirror . mirror) Nodo x l r
```
$\equiv \langle def.\circ \rangle$
```
mirror (mirror (Nodo x l r))
```
$\equiv \langle def.mirror.2 \rangle$
```
mirror (Nodo x (mirror r) (mirror l))
```
$\equiv \langle def.mirror.2 \rangle$
```
Nodo x (mirror (mirror l)) (mirror (mirror r))
```
$\equiv \langle def.\circ \rangle$
```
Nodo x ((mirror . mirror) l) ((mirror . mirror) r)
```
$\equiv \langle H.I.; def.id \rangle$
```
Nodo x l r
```
$\equiv \langle def.id \rangle$
```
id (Nodo x l r)
```

e)

- Caso base `Hoja x`:

  `hojas (Hoja x)`

  $\equiv \langle def.hojas.1 \rangle$

  `1`

  $<$

  `2^1`

  $\equiv \langle def.altura.1 \rangle$

  `2^(altura (Hoja x))`

- Caso inductivo `Nodo x l r`:

  `hojas (Nodo x l r)`

  $\equiv \langle def.hojas.2 \rangle$

  `hojas l + hojas r`

  $< \langle H.I. \rangle$

  `2^(altura l) + 2^(altura r)`

  $<$

  ``2 * 2^(altura l `max` altura r)``

  $\equiv$

  ``2^(1 + (altura l `max` altura r))``

  $\equiv \langle def.altura.2 \rangle$

  `2^(altura (Nodo x l r))`

9. Dadas las siguientes definiciones:

```haskell
data AGTree a = Node a [AGTree a]
ponerProfs n (Node x xs) = Node n (map (ponerProfs (n+1)) xs)
```

   a) Definir una función `alturaAGT` que calcule la altura de un `AGTree`.

   b) Definir una función `maxAGT` que dado un `AGTree` de enteros devuelva su mayor elemento.

   c) Demostrar que `maxAGT . (ponerProfs 1)` $\equiv$ `alturaAGT`.

Soluciones

a)
```
alturaAGT (Node _ []) = 1
alturaAGT (Node _ xs) = 1 + maximum (map alturaAGT xs)
```

b)
```
maxAGT (Node x []) = x
maxAGT (Node x xs) = maximum (x:(map maxAGT xs))
```

c)

- Caso base `Node x []`:

  `(maxAGT . (ponerProfs 1)) Node x []`

  $\equiv \langle def.\circ; def.ponerProfs \rangle$

  `maxAGT (Node 1 (map (ponerProfs 2) []))`

  $\equiv \langle def.map.1 \rangle$

  `maxAGT (Node 1 [])`

  $\equiv \langle def.maxAGT.1 \rangle$

  `1`

  $\equiv \langle def.alturaAGT.1 \rangle$

  `alturaAGT (Node x [])`

- Lema 1 `ponerProfs (n+1)` $\equiv$ `mapT (+1) . ponerProfs n`: COMPLETAR.
- Lema 2 `maxAGT . mapT (+1)` $\equiv$ `(+1) . maxAGT`: COMPLETAR.
- Lema 3 `alturaAGT t` $\geq$ `1`: COMPLETAR.

- Caso inductivo `Node x xs`:

  `(maxAGT . (ponerProfs 1)) (Node x xs)`

  $\equiv \langle def.\circ; def.ponerProfs.1\rangle$

  `maxAGT (Node 1 (map (ponerProfs 2) xs))`

  $\equiv \langle def.maxAGT.2\rangle$

  `maximum (1:(map maxAGT (map (ponerProfs 2) xs)))`

  $\equiv \langle prop.map\rangle$

  `maximum (1:(map (maxAGT . (ponerProfs 2)) xs))`

  $\equiv \langle lema.1\rangle$

  `maximum (1:(map (maxAGT . mapT (+1) . ponerProfs 1) xs))`

  $\equiv \langle lema.2\rangle$

  `maximum (1:(map ((+1) . maxAGT . ponerProfs 1) xs))`

  $\equiv \langle prop.map\rangle$

  `maximum (1:(map (+1) (map (maxAGT. ponerProfs 1) xs)))`

  $\equiv \langle def.map; H.I.\rangle$

  `maximum (1:(map (+1) (map alturaAGT xs)))`

  $\equiv \langle lema.3\rangle$

  `maximum (map (+1) (map alturaAGT xs))`

  $\equiv \langle prop\rangle$

  `1 + maximum (map alturaAGT xs)`

  $\equiv \langle def.alturaAGT.2\rangle$

  `alturaAGT (Node x xs)`

10. Dadas las siguientes definiciones:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)

flatten (Leaf x)     = [x]
flatten (Node x l r) = flatten l ++ [x] ++ flatten r

mapTree f (Leaf x)     = Leaf (f x)
mapTree f (Node x l r) = Node (f x) (mapTree f l) (mapTree f r)
```

demostrar que `(map f) . flatten` $\equiv$ `flatten . (mapTree f)`.

Solución

- Caso base `Leaf x`:

  `((map f) . flaten) (Leaf x)`

  $\equiv \langle def.\circ \rangle$

  `map f (flaten (Leaf x))`

  $\equiv \langle def.flatten.1 \rangle$

  `map f (x:[])`

  $\equiv \langle def.map.2 \rangle$

  `(f x) : (map f [])`

  $\equiv \langle def.map.1 \rangle$

  `(f x) : []`

  $\equiv \langle def.flatten.1 \rangle$

  `flatten (Leaf (f x))`

  $\equiv \langle def.mapTree.1 \rangle$

  `flatten (mapTree f (Leaf x))`

  $\equiv \langle def.\circ \rangle$

  `(flatten . (mapTree f)) (Leaf x)`

- Caso inductivo `Node x l r`:

  `((map f) . flaten) (Node x l r)`

  $\equiv \langle def.\circ; def.flatten.2 \rangle$

  `map f (flaten l ++ [x] ++ flatten r)`

  $\equiv \langle prop.map \rangle$

  `map f (flaten l) ++ map f [x] ++ map f (flatten r)`

  $\equiv \langle def.\circ; H.I. \rangle$

  `(flatten . (mapTree f)) l ++ map f [x] ++ (flatten . (mapTree f)) r`

  $\equiv \langle def.flatten.2; def.map; def.\circ \rangle$

  `flatten (Node (f x) (mapTree f l) (mapTree f r))`

  $\equiv \langle def.\circ; def.mapTree.2 \rangle$

  `(flatten . (mapTree f)) (Node x l r)`

11. Dada la siguiente definición,

```
join []       = xs
join (xs:xss) = xs ++ join xss
```

demostrar:

   a) `join` $\equiv$ `concat . (map id)`
   b) `join . join` $\equiv$ `join . (map join)`

Soluciones

   a) COMPLETAR.
   b) COMPLETAR.

12. Dadas las funciones `insert :: Ord a => a -> Bin a -> Bin a`, que agrega un elemento a un BST dado, y `inorder :: Ord a => Bin a -> [a]`, que realiza un recorrido *inorder* sobre un BST, dadas en clase de teoría, probar las siguientes propiedades sobre las funciones:

   a) Si t es un BST, entonces `insert x t` es un BST.
   b) Si t es BST, entonces `inorder t` es una lista ordenada.

Soluciones

   a) COMPLETAR.
   b) COMPLETAR.

13. Dadas las definición de funciones que implementan *leftist heaps*, dadas en clase, probar que si `h1` y `h2` son leftist heaps, entonces `merge h1 h2` es un leftist heap.

Solución

```
checkH E             = True
checkH (N _ _ l r)  = rank l >= rank r && checkH l && checkH r
```

- Lema: Si `h1` y `h2` son leftist heaps, entonces `check (makeH h1 h2)` también lo es.
  - Caso base `E`:

    `checkH (makeH x E h2)`

    $\equiv \langle def.makeH \rangle$

    `checkH (if rank E >= rank h2 then N (rank h2 + 1) x E h2 else N (x + 1) x h2 h1)`

    $\equiv \langle def.rank.1 \rangle$

    `checkH (if 0 >= rank h2 then N (rank h2 + 1) x E h2 else N (rank E + 1) x h2 E)`

    - Caso $0 \geq$ `rank h2`

      $\equiv \langle def.if \rangle$

      `checkH (N (rank h2 + 1) x E h2)`

      $\equiv \langle def.checkH.2 \rangle$

      `rank E >= rank h2 && checkH E && checkH h2`

      $\equiv \langle def.rank.1 \wedge def.checkH.1 \wedge Hipotesis \rangle$

      `0 >= rank h2 && True && True`

      $\equiv \langle caso.0 \geq rank.h2 \wedge def.\&\& \rangle$

      `True`

    - Caso $0 <$ `rank h2`

      $\equiv \langle def.if \rangle$

      `checkH (N (rank E + 1) x h2 E))`

      $\equiv \langle def.checkH.2 \rangle$

      `rank h2 >= rank E && checkH h2 && checkH E`

      $\equiv \langle def.rank.1 \wedge Hipotesis \wedge def.checkH.1 \rangle$

      `rank h2 >= 0 && True && True`

      $\equiv \langle caso.0 < rank.h2 \wedge def.\&\& \rangle$

      `True`

- Caso inductivo `N x a1 b1`:
  `checkH (makeH x (N x a1 b1) h2)`
  $\equiv \langle def.makeH + def.rank.2 \rangle$
  `checkH (if x >= rank h2 then N (y + 1) x (N x a1 b1) h2 else N (x + 1) x h2 (N x a1 b1))`
  - Caso $x \geq$ `rank h2`
    `checkH (N (y + 1) x (N x a1 b1) h2)`
    $\equiv \langle def.checkH.2 + def.rank.1 \rangle$
    `x >= rank h2 && checkH (N x a1 b1) && checkH h2`
    $\equiv \langle x \geq rank.h2 \wedge Hipotesis \rangle$
    `True && checkH (N x a1 b1) && True`
    $\equiv \langle def.checkH \wedge H.I. \wedge def.\&\& \rangle$
    `True`
  - Caso $x <$ `rank h2`: COMPLETAR.

- `checkH (merge h1 h2)` $\equiv$ `True`.
  - Caso base: COMPLETAR.
  - Caso inductivo:
    `checkH (merge (N x a1 b1) h2)`
    $\equiv \langle def.merge \rangle$
    `checkH (if x <= rank h2 makeH x a1 (merge b1 h2) else makeH rank h2 a2 (merge h1 b2))`
    - Caso $x \leq$ `rank h2`:
      `checkH (makeH x a1 (merge b1 h2))`
      $\equiv \langle Hipotesis \wedge H.I \wedge Lema \rangle$
      `True`
    - Caso $x >$ `rank h2`: COMPLETAR.

14. Dar el principio de inducción estructural para el siguiente tipo de datos:
    `data F = Zero | One F | Two (Bool -> F)`

    Solución Sea P una propiedad definida sobre elementos de F.

    - Si P vale en `Zero` y ademas;
    - si P vale en `f` también vale en `One f` y ademas;
    - si para cualquier `g`, P vale en `g True` y `g True` entonces también vale en `Two (g True)` y `Two (g False)`;

    entonces P vale para todos los elementos de F.