Tipos Algebraicos

Taller de Álgebra I

Primer cuatrimestre de 2016

Recordemos algunos tipos que fuimos creando en clases anteriores

¿Cuál es la diferencia entre type y data?

Tipos Enumerados usando data.

```
data Dia = Lunes | Martes | Miercoles | ... | Sabado | Domingo
```

También algunos renombres de tipos con type.

```
data Direccion = Norte | Sur | Este | Oeste deriving Show
type Pos = (Int,Int)
type Tortuga = (Pos,Direccion)
```

Tipos Paramétricos

los constructores toman parámetros

```
data Figura = Rectangulo Punto Punto | Circulo Punto Float
    deriving (Eq, Show)

type Punto = (Float, Float)
```

Por ejemplo, podemos decir que:

```
Rectangulo (0.3, 0.5) (0.3, 0.5) :: Figura
Circulo (0,0+20.23) 88.3 :: Figura
area (Rectangulo (0,0) (1,0)) :: Float
```

Tipos Algebraicos Recursivos

Algunos constructores toman parámetros, y al menos uno de ellos es del mismo tipo que se está definiendo.

Árboles Estrictamente Binarios de Enteros

Un árbol estrictamente binario de enteros es una estructura muy utilizada en computación y matemática. Veamos cómo podemos representarlos:

```
data Arbol = Hoja Integer | Ramificacion Arbol Integer Arbol
```

En este caso, `Hoja` es un constructor que toma 1 parámetro y `Ramificacion` es un constructor que toma 3 parámetros, de los cuales 2 son del tipo que se está definiendo.

Ejercicios

Determinar el tipo de las siguientes expresiones:

- `Hoja 10`
- `Ramificacion (Hoja 20) 10 (Hoja 30)`
- `[Ramificacion (Hoja 2) 5 (Ramificacion (Hoja 1) 10 (Hoja 0)), Hoja 0]`
- `Ramificacion (Hoja 3) 5 (Ramificacion (Hoja 1))`

Más sobre los árboles

Árboles Estrictamente Binarios de Enteros

```
data Arbol = Hoja Integer | Ramificacion Arbol Integer Arbol
```

Si lo necesitan, puede agregar `deriving (Eq, Show)` al final de la definición.

Ejercicios

Implementar las siguientes funciones:

- `esHoja :: Arbol -> Bool` que determina si un árbol es o no una hoja.
- `sumaNodos :: Arbol -> Integer` que devuelve la suma de los valores del árbol.
- `altura :: Arbol -> Integer` que devuelve la altura de un árbol.
- `pertenece :: Integer -> Arbol -> Bool`
  que indica si un elemento pertenece o no a un árbol.
- dado el tipo `data Dir = Der | Izq`
  implementar `busqueda :: [Dir] -> Arbol -> Integer` que recorre el árbol siguiendo la lista de instrucciones y devuelve el valor que se encuentre luego de recorrerlo (asumir que la lista lleva a un elemento y no se termina el árbol antes de encontrarlo)

Todo muy lindo... peeero

¿Y si quisiera que mis árboles contengan Chars?

Árboles Estrictamente Binarios de Chars

```
data ArbolChar = HojaC Char | RamifC ArbolChar Char ArbolChar
```

Ahora quiero árboles de tuplas de la pinta (Usuario, Clave).

Árboles Estrictamente Binarios de Tuplas (Usuario, Clave)

```
type Usuario = String
type Clave = String
type UC = (Usuario, Clave)
data ArbolUC = HojaUC UC | RamifUC ArbolUC UC ArbolUC
```

¿Se podrá generalizar este comportamiento?

Tipos genéricos

¿Y qué tal si usamos tipos genéricos?

```
data Arbol t = Hoja t | Ramif (Arbol t) t (Arbol t)
```

Estamos definiendo infinitos tipos (uno por cada posible tipo `t`).

Ejemplos

- `Hoja 20 :: Arbol Integer`
- `Ramif (Hoja 10) 2 (Hoja 10) :: Arbol Integer`
- `Ramif (Hoja 'b') 'a' (Hoja 'c') :: Arbol Char`
- `Ramif (Hoja "10") "Algebra" (Hoja "10") :: Arbol String`
- `Ramif (Ramif (Hoja 10) 2 (Hoja 10)) 6 (Hoja 10) :: Arbol Integer`
- `Ramif (Hoja (Hoja 10)) (Ramif (Hoja 10) 2 (Hoja 10)) (Hoja (Hoja 10)) :: Arbol (Arbol Integer)`

Ejercicios

Implementar las siguientes funciones:

1. `esHoja :: Arbol a -> Bool`
   que determina si el árbol es una hoja.
2. `maximo :: Ord a => Arbol a -> a`
   que devuelve el máximo elemento de un árbol de elementos con orden.
3. `raiz :: Arbol a -> a`
   que devuelve el valor del nodo principal del árbol.
4. `todosIguales :: Eq a => Arbol a -> Bool`
   que determina si todos los nodos del árbol tienen el mismo valor.
5. `espejar :: Arbol a -> Arbol a`
   que invierte el árbol de manera que esté espejado.
6. `esHeap :: Ord a => Arbol a -> Bool`
   que valga verdadero en un árbol si cada nodo (salvo la raíz) es mayor o igual que su padre.

Ejercicios adicionales

Suponiendo que no existen las listas en *Haskell*, definiremos nuestras propias listas usando tipos algebraicos:

```
data Lista a = Vacia | Agregar a (Lista a)
```

Implementar las siguientes funciones:

1. `vacia :: Lista a -> Bool` que determina si una lista es o no la lista vacía.
2. `suma :: Lista Float -> Float` que determina la suma de una lista de floats.
3. `enPosicion :: Lista a -> Integer -> a` que devuelve el elemento en la posición pasado como parámetro.
4. `iguales :: Eq a => Lista a -> Lista a -> Bool`
   que determina si dos listas de elementos (comparables por igual) son iguales.
5. `juntar :: Lista a -> Lista b -> Lista (a,b)`
   que crea una lista resultante de formar tuplas con los elementos de cada lista. Se asume que las dos listas tienen la misma longitud. Por ejemplo:
   `juntar (Agregar 1 (Agregar 2 Vacia)) (Agregar 'a' (Agregar 'b' Vacia))` $\rightsquigarrow$
   `Agregar (1, 'a') (Agregar (2, 'b') Vacia)`
6. Hacer que `Lista a` sea instancia de `Show` implementando la función correspondiente para que, por ejemplo `Agregar 2 (Agregar 3 Vacia)` se vea en pantalla como `[2,3]`.