Tipos paramétricos y recursivos

Taller de Álgebra I

Segundo cuatrimestre de 2016

Tipos paramétricos

Podemos hacer que los constructores de un tipo de datos reciban **parámetros**. A los tipos construidos de esta forma los llamamos **tipos paramétricos**.

Por ejemplo:

```
data Vector = Vector2D Float Float | Vector3D Float Float Float
```

Igual que antes, podemos hacer *pattern matching* utilizando los constructores del tipo.

```
norma Vectorial :: Vector -> Float
norma Vectorial Vector2D x y   = sqrt (x ^ 2 + y ^ 2)
norma Vectorial Vector3D x y z = sqrt (x ^ 2 + y ^ 2 + z ^ 2)
```

El tipo de datos `Figura`

Consideremos el siguiente tipo de datos, que representa puntos en el plano.

```
data Punto = Punto2D Float Float
```

Queremos utilizarlo para definir un tipo de datos, `Figura`, que nos permita representar tanto rectángulos como círculos.
Para simplificar, consideraremos rectángulos con lados paralelos a los ejes de coordenadas.

- **¿Qué información necesitamos para representar de forma unívoca un rectángulo?**
  Idea: dos `Punto`, representando a un par de esquinas opuestas.
- **¿Y para representar un círculo?**
  Idea: un `Punto` para representar el centro y un `Float` para representar el radio.

Usando estas ideas:

```
data Figura = Rectangulo Punto Punto | Circulo Punto Float
```

Ejercicio: Algunas funciones con `Figura`

```
data Punto = Punto2D Float Float
data Figura = Rectangulo Punto Punto | Circulo Punto Float
```

Ejercicios

Definir las siguientes funciones:

- `circuloUnitario :: Figura`,
  que devuelve un círculo de radio 1 centrado en el origen.
- `cuadrado :: Float -> Figura`,
  que devuelve un cuadrado con una esquina en el origen, tomando por parámetro la longitud de su diagonal.
- `perimetro :: Figura -> Float`,
  que recibe una figura y calcula su perímetro.
- `area :: Figura -> Float`,
  que recibe una figura y calcula su área.

Tipos paramétricos genéricos

Con los tipos paramétricos, podemos representar un par ordenado de números enteros:

```
data ParOrdenado = Par Integer Integer
```

Pero también podríamos querer pares de elementos de otros tipos. Por ejemplo:

```
data ParOrdenado = Par Float Float
data ParOrdenado = Par Bool Bool
```

¿Podremos generalizar esto?

Sí. Podemos usar **tipos paramétricos genéricos**, y definir toda una familia de tipos en una sola línea.

```
data ParOrdenado a = Par a a
```

¿Hace falta que los dos elementos sean del mismo tipo?

No. Esto también vale:

```
data ParOrdenado a b = Par a b
```

Tipos paramétricos genéricos: Ejercicios

Ejercicios

Considerando la definición

```
data ParOrdenado a b = Par a b
```

- ¿Cuál es el tipo de las siguientes expresiones?
  - `Par 4 5.25`
  - `Par False [1,8,-4]`
  - `Par (Par 'a' 5) 3.14`
- Programar las siguientes funciones:
  - `primero :: ParOrdenado a b -> a`,
    que devuelve el primer elemento de un par ordenado.
  - `segundo :: ParOrdenado a b -> b`,
    que devuelve el segundo elemento de un par ordenado.

El secreto de las tuplas

Todo lo anterior tiene un aire familiar... ¿No se parece mucho a las **tuplas**?

Efectivamente, las tuplas que Haskell trae "de fábrica" son tipos paramétricos genéricos. Sus constructores son los ya conocidos paréntesis y comas. Así, tenemos:

- `(,) :: a -> b -> (a, b)`,
- `(,,) :: a -> b -> c -> (a, b, c)`,
- `(,,,) :: a -> b -> c -> d -> (a, b, c, d)`, etc.

Esto es lo que nos permite hacer *pattern matching* en tuplas.

La estructura recursiva de las listas

Con las herramientas que vimos hasta ahora, podemos construir varios de los tipos de datos con los que venimos trabajando desde el principio del taller. ¿Qué pasa con las **listas**?

Una característica fundamental de las listas es que pueden definirse de manera **recursiva**.

Toda lista es:

(a) la **lista vacía**, o

(b) el resultado de **agregar un elemento** al principio de **otra lista**.

Tipos de datos recursivos

Tipos recursivos

Decimos que un tipo de datos es **recursivo** cuando alguno de sus constructores toma por parámetro una instancia del *mismo tipo* que estamos definiendo.

Como ya vimos, las listas tienen *estructura recursiva*, por lo que podemos definirlas en Haskell como un tipo recursivo.

Por ejemplo, podríamos definir nuestras propias listas de esta forma:

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

Tipos recursivos: Ejercicios

Ejercicios

A partir de la siguiente definición de listas como tipos recursivos

```
data Lista a = ListaVacia | Agregar a (Lista a) deriving (Show)
```

programar las siguientes funciones:

1. `esVacia :: Lista a -> Bool`
   que determina si una lista es o no la lista vacía.
2. (a) `cabeza :: Lista a -> a`,
   (b) `cola :: Lista a -> Lista a`, y
   (c) `concatenar :: Lista a -> Lista a -> Lista a`,
   (d) `longitud :: Lista a -> Integer`,
   equivalentes a las funciones `head`, `tail`, `(++)` y `length` de las listas por defecto de Haskell.
3. `suma :: Lista Float -> Float`
   que determina la suma de una lista de `Float`.
4. `posicion :: Lista a -> Integer -> a`
   que devuelve el elemento en la posición pasado como parámetro.

El regreso de las clases de tipos

La clase pasada, vimos como hacer que nuestros tipos formen parte de algunas clases de tipos, como `Show`, `Eq` u `Ord`. Para esto, usamos la palabra clave `deriving`; así, Haskell define de forma automática las funciones necesarias.

¿Cómo hacemos para definir **nosotros mismos** estas funciones? Usamos la palabra clave `instance`.

Un ejemplo sencillo: si tenemos el tipo de datos

```
data Bool = True | False
```

en vez de usar `deriving Show`, podemos hacer lo siguiente:

```
instance Show Bool where
    show True  = "True"
    show False = "False"
```

Para pensar entre todos

¿En qué casos resulta interesante definir nuestras propias funciones para `Show`, `Eq` u `Ord`, en lugar de utilizar las que Haskell infiere por defecto?

Clases de tipos: Ejercicios

Ejercicios

1. Definir una función `sonIguales :: Eq a => Lista a -> Lista a -> Bool`, que determine si dos listas (cuyos elementos sean comparables por igualdad) son o no iguales. ¡No se puede usar `deriving Eq`!
2. Utilizar la función anterior para hacer que el tipo `Lista a` sea instancia de `Eq`, **siempre y cuando** el tipo a sea instancia de `Eq`.
   Para hacer esto, en vez de `instance Eq (Lista a)`, tendremos que escribir `instance Eq a => Eq (Lista a)`.
3. Hacer que `Lista a` sea instancia de `Show`, siempre y cuando a sea también instancia de `Show`. La forma de mostrar las listas por pantalla deberá ser la que usa Haskell en sus listas por defecto.
   Por ejemplo, `Agregar 2 (Agregar 3 Vacia)` debería verse en pantalla como `[2,3]`.

Recordar:

- Para que un tipo `t` sea instancia de `Eq`, hay que definir una función `(==) :: t -> t -> Bool`.
- Para que un tipo `t` sea instancia de `Show`, hay que definir una función `show :: t -> String`.

Pattern matching en listas

Las listas por defecto de Haskell también están definidas como tipos recursivos. Sus constructores son los siguientes:

- `[] :: [a]`, la lista vacía, y
- `(:) :: a -> [a] -> [a]`, que agrega un elemento al principio de otra lista.

Utilizando estos constructores, podemos hacer *pattern matching* en listas.

Por ejemplo, la función:

```
producto :: [Integer] -> Integer
producto xs | length xs == 0 = 1
            | otherwise      = head xs * producto (tail xs)
```

puede reescribirse como:

```
producto :: [Integer] -> Integer
producto []     = 1
producto (x:xs) = x * producto xs
```

Ejercicios

1. Completar las siguientes definiciones.

   (a)
   ```
   longitud :: [a] -> Integer
   longitud [] =
   longitud (x : []) =
   longitud (x : y : []) =
   longitud (x : y : z : []) =
   longitud ( _ : _ : _ : xs) =
   ```

   (b)
   ```
   iniciales :: [Char] -> [Char] -> [Char]
   iniciales nombre apellido =
       where (n : _) = nombre
             (a : _) = apellido
   ```

2. Implementar las siguientes funciones, **utilizando pattern matching**.

   (a) `tuplas :: [a] -> [b] -> [(a,b)]`,
   que calcule una lista de tuplas donde las primeras componentes sean elementos de la primera lista, y las segundas componentes de la segunda lista. Si las listas son de distinta longitud, se deben usar tantos elementos como se pueda.
   Nota: en Haskell esta función existe, y se llama `zip`.
   Ejemplo: `tuplas [1,2,3,4] ['a','b','c']` $\rightsquigarrow$ `[(1,'a'),(2,'b'),(3,'c')]`

   (b) `intercalar :: [a] -> [a] -> [a]`,
   que dadas dos listas, devuelve la lista resultante de colocar de forma intercalada un elemento de cada lista.
   Ejemplo: `intercalar [1,3,5,7,9] [2,4]` $\rightsquigarrow$ `[1,2,3,4,5,7,9]`