Universidad de Huelva

Práctica 4

Definición de tipos y clases

Índice

Índice

- La instrucción *type* permite asignar un alias a una declaración de tipo
- La sintaxis es la siguiente

> **typeDecl ::= type *simpleType* = *expresion_de_tipo***

- El símbolo *simpleType* define el identificador del tipo y puede incluir variables de tipo.

> **simpleType ::= Identificador $tvar_1$ $tvar_2$ ... $tvar_n$**

- Las variables incluidas en *simpleType* deben corresponder a las variables de tipo incluidas en la expresión.

- Por ejemplo, el tipo *String* es en realidad un sinónimo de lista de caracteres. De esta forma podemos describir las funciones definidas sobre cadenas en un estilo más sencillo.

```
type String = [Char ]
words :: String -> [String ]
```

- Utilizando variables de tipo podemos definir el tipo *Pair* como un sinónimo de una 2-tupla.

```
type Pair a b = ( a , b )
```

Índice

- La forma convencional de definir un nuevo tipo de dato (lo que se conoce como ***datos algebraicos***) es por medio de la instrucción *data*.
- La sintaxis es la siguiente

dataDecl ::= data [*context* =>] *simpleType* [= *constrs*] [*deriving*]

simpleType ::= Identificador $tvar_1$ $tvar_2$ ... $tvar_n$

- El contexto es opcional y permite indicar las clases a las que pertenecen los variables de tipo.

- La declaración de tipos de datos algebraicos se basa en definir constructores del tipo.
- Por ejemplo, el tipo de dato *Bool* está basado en dos constructores: *True* y *False*. Los constructores de tipo utilizan identificadores que comienzan en mayúscula.

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

- Los constructores sin argumentos permiten definir fácilmente enumeraciones:

```
data Dia = Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo
```

- Los constructores pueden tener argumentos. No es necesario que todos los constructores de un tipo tengan los mismos argumentos.
- Por ejemplo, podemos definir un tipo de dato que represente una figura geométrica (Shape) que pueda tomar dos formas: un círculo o un rectángulo. El círculo necesita la posición del centro y el radio mientras que el rectángulo necesita la posición de una esquina y su contraria.

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
```

- Los constructores de tipo son funciones que toman los parámetros como entrada y generan un valor del tipo descrito

- Los constructores de tipo son funciones que toman los parámetros como entrada y generan un valor del tipo descrito.

```
Prelude> :type  Circle
Circle  ::  Float -> Float -> Float -> Shape
```

- Podemos utilizar los constructores para definir procesos de emparejamiento (pattern matching).

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs (x2 - x1)) * (abs (y2 - y1))
```

- El constructor de un tipo puede tener el mismo identificador que el tipo . De hecho esto es una elección común en tipos de datos que solo utilizan un único constructor.
- Por ejemplo, podemos definir un punto como un tipo *Point* y utilizarlo en las definiciones de figuras.

```
data  Point  =  Point  Float  Float
data  Shape  =  Circle  Point  Float  |  Rectangle  Point  Point


surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) =  (abs (x2 - x1))
                                                    * (abs (y2 - y1))
```

- Los tipos de datos algebraicos permiten definir estructuras de tipo registro.
- Por ejemplo, podríamos definir el tipo Persona que almacenara el nombre, apellidos y edad. Para acceder a estos cammpos podríamos definir funciones basadas en patrones.

```
data Persona = Persona String String Int
nombre:: Persona -> String
nombre (Persona n _ _ ) = n
apellidos :: Persona -> String
apellidos (Persona _ a _) = a
edad :: Persona -> Int
edad (Persona _ _ e) = e
```

- Existe una sintaxis especial que permite definir las estructuras de tipo registro de una forma mucho más compacta.

```
data Persona = Persona { nombre:: String
                       , apellidos :: String
                       , edad :: Int }
```

- Este formato define directamente las funciones asociadas a los campos y permite indicar el valor de cada campo al construir el dato.

```
Persona { nombre= "Juan" , apellidos = "Nadie" , edad = 50 }
```

- El orden de los campos puede incluso ser diferente al de la definición del tipo.

- Los tipos de datos pueden ser instancias de clases.
- Para indicar que un tipo desarrolla una determinada clase se puede utilizar la clausula *deriving* al final de la definición.

```
data Point  =  Point  Float  Float  deriving (Eq, Show)
data Persona = Persona { nombre:: String
                       , apellidos :: String
                       , edad :: Int } deriving (Eq, Show)
```

- De esta forma, el compilador de Haskell intenta inferir el comportamiento del tipo a partir del comportamiento de los tipos de datos que aparecen en la definición.

- Las definiciones de tipos pueden ser recursivas, es decir, las expresiones de tipos de los constructores pueden utilizar el mismo tipo de dato que estamos definiendo.
- Por ejemplo, podemos definir un tipo de dato árbol con dos tipos de nodos: un nodo hoja que almacene un valor entero y un nodo interno que almacene dos ramas.

```
data Arbol = Hoja Int
           | Nudo Arbol Arbol
           deriving(eq, Show)

Nudo (Nudo (Hoja 1) (Hoja 2)) Hoja 3
```

- Las definiciones de tipos pueden incluir parámetros de tipo.
- Por ejemplo, supongamos que queremos desarrollar una función de búsqueda en una lista que devuelva el primer elemento que cumpla una condición. ¿Cómo podemos expresar el caso de que no se encuentre ningún elemento? Para estos casos se suele utilizar un tipo de dato predefinido en Prelude llamado *Maybe*.

```
data Maybe a = Nothing
             | Just a
```

- El tipo *Maybe* está definido con un parámetro de tipo. Para tener un tipo de dato concreto debemos indicar, por ejemplo, *Maybe Int*.

-

-

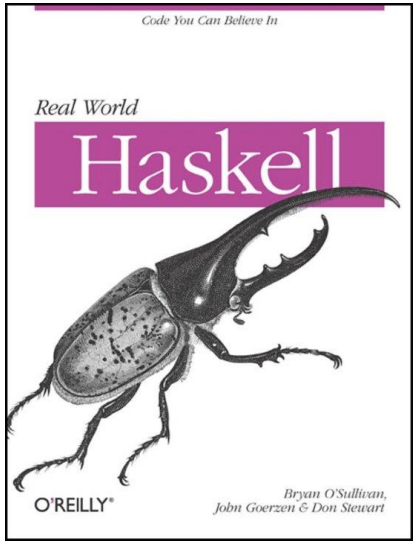
-

- Puede encontrarse una descripción muy didáctica del sistema de tipos y clases de Haskell en el capítulo 8 del manual "*Aprende Haskell por el bien de todos*".

http://aprendehaskell.es/content/ClasesDeTipos.html

Ejercicios:

- Probar en el intérprete de Haskell los diferentes ejemplos incluidos en los capítulos "Defining Types, Streamlining Functions" y "Using typeclasses" del libro "Real World Haskell"



http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html