Prácticas de laboratorio de LTP
(Parte II : Programación Funcional)

Práctica 6: Módulos y Polimorfismo en Haskell (II)

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Jose Luis Pérez
jlperez@dsic.upv.es

Introducción

Esta nueva práctica también se dividirá en dos sesiones. Veremos como **importar módulos** indicando que parte de cada módulo el visible o no. Y seguiremos profundizando en los **mecanismos de polimorfismo** de Haskell ya estudiados en las prácticas anteriores, a los que añadiremos las **clases de tipos**: como definirlos y crear instancias de clases ya definidas.

Nota: En Poliformat se dispone de un enlace a un libro de Haskell en castellano. Y también el fichero **codigoEnPdf_P6** que podéis utilizar para copiar y pegar los ejemplos que se presentan durante la sesión.

2.2. Polimorfismo ad hoc o sobrecarga: Sin clases de tipos

Para definir una función cuyo comportamiento dependa del tipo de valor recibido no hace falta necesariamente recurrir a clases de tipos, como se muestra en el siguiente ejemplo:

```
module Shape1 where
  type Side = Float
  type Apothem = Float
  type Radius = Float

  data Shape = Pentagon Side Apothem |
               Circle Radius deriving (Eq, Show)

  perimeter :: Shape -> Float
  perimeter (Pentagon s a) = 5 * s
  perimeter (Circle r) = 2 * pi * r
```

```
import Shape1
main = do
   let f = (Pentagon 5 4)
   putStrLn ("perimeter is " ++ show (perimeter f))
```

Basta con definir un tipo algebraico, **Shape**, con diferentes constructores para los diferentes tipos de figuras.

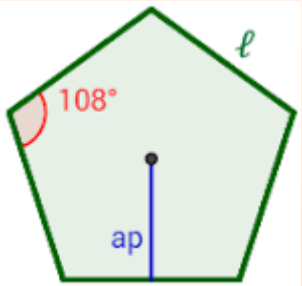
La cabecera indica que la función está definida para todos los constructores de **Shape**. El comportamiento de la función **perimeter** dependerá del argumento de entrada que reciba (el tipo de **Shape: Pentagon** o **Circle**).

Ejercicio 6: Modificar este tipo **Shape** para que tenga también una función **area** que devuelva el área de una figura (que solamente puede ser **Pentagon** o **Circle**).

2.2. Polimorfismo ad hoc o sobrecarga: Sin clases de tipos

Recordemos para resolver este ejercicio que:

Àrea círculo = 2*pi^2 y
Àrea pentagono = 5*lado*apotema/2



Definida la función **area** (módulo **Shape1**) , considérese la siguiente función **volumePrism** que calcula el volumen de un prisma cuya base es una figura:

```
type Height = Float
type Volume = Float

volumePrism :: Shape -> Height -> Volume
volumePrism base height = (area base) * height
```

La función **volumePrism** es capaz de utilizar un elementos de tipo **Shape**, definidos mediante los constructores **Pentagon** y **Circle** e invocar a la función **area**, que ejecutará una función **area** u otra dependiendo del tipo. Se dirá que la funcián **area** tiene polimorfismo **ad hoc**.

2.2. Polimorfismo ad hoc o sobrecarga: Con clases de tipos

El problema de no utilizar clases de tipos es que no resulta posible añadir dinámicamente más constructores para el tipo **Shape**. La forma de solucionarlo es definir una **clase de tipos Shape** y después tantas instancias de ella como figuras concretas se quieran crear:

```
module Shape2 where
  type Side = Float
  type Apothem = Float
  type Radius = Float

  class Shape a where
    perimeter :: a -> Float

  data Pentagon = Pentagon Side Apothem deriving Show
  data Circle = Circle Radius deriving Show

  instance Shape Pentagon where
    perimeter (Pentagon s a) = 5 * s

  instance Shape Circle where
    perimeter (Circle r) = 2 * pi * r
```

Ahora **Shape** es una clase de tipos en vez de un tipo algebraico. Las instancias de la clase **Shape** deben definir la función **perimeter**. La variable de tipo **a** representa a las instancias de **Shape**.

En vez de un único tipo algebraico podemos utilizar un tipo algebraico para cada tipo de figura (cada tipo tiene su propio constructor)

Tanto **Pentagon** como **Circle** se declaran como instancias de **Shape**. Deben implementar la función **perimeter**.

2.2. Polimorfismo ad hoc o sobrecarga: Con clases de tipos

Si ahora quisiéramos añadir un nuevo tipo de figura, podríamos hacerlo en un nuevo fichero (**testShape.hs**) sin tener que modificar el módulo **Shape2**:

```
import Shape2
type Base = Float
type Height = Float
data Rectangle = Rectangle Base Height deriving Show

instance Shape Rectangle where
    perimeter (Rectangle b h) = 2 * b + 2 * h

main = do
   let f1 = (Pentagon 5 4)
       f2 = (Circle 5)
       f3 = (Rectangle 5 4)
   putStrLn ("perimeter" ++ show f1 ++ " is " ++ show (perimeter f1))
   putStrLn ("perimeter" ++ show f2 ++ " is " ++ show (perimeter f2))
   putStrLn ("perimeter" ++ show f3 ++ " is " ++ show (perimeter f3))
```

2.2. Polimorfismo ad hoc o sobrecarga: Con clases de tipos

Ejercicio 7: Modificar esta clase de tipos **Shape** (módulo **Shape2**) para que tenga también una función **area** que devuelva el área de una figura. Para ello, modifíquense adecuadamente las instancias de las clases **Pentagon** y **Circle**.

Definida la función **area**, considérese la siguiente función **volumePrism** que calcula el volumen de un prisma cuya base es una figura (instancia de **Shape** a):

```
type Height = Float
type Volume = Float

volumePrism :: (Shape a) => a -> Height -> Volume
volumePrism base height = (area base) * height
```

La función **volumePrism** es capaz de utilizar un elemento de la clase **Shape** a, en concreto términos de los tipos **Pentagon** y **Circle** (instancias de **Shape** a) e invocar a la función **area**, que ejecutará una función **area** u otra dependiendo del tipo. Se dirá que la funcián **area** tiene polimorfismo **ad hoc**.

2.2. Polimorfismo ad hoc o sobrecarga: Con clases de tipos

Ejercicio 8: Añadir a la clase de tipos **Shape** la función **volumePrism** y definir una nueva función **surfacePrism** que calcule la superficie de un prisma.

Ejercicio 9: Modificar la definición basada en clases de tipos para que sea posible mostrar y comparar (igualdad) valores de la clase de tipos **Shape** instanciando las clases de tipos **Show** y **Eq**. La idea es básicamente reemplazar la línea:

```
class Shape a where
```

por

```
class (Eq a, Show a) => Shape a where
```

Se obliga a que las instancias de **Shape** también lo sean de **Eq** y **Show**.

y luego incluir el código necesario para que compile y funcione correctamente.

Ejemplo: Instancias de las clases de tipos **Eq** y **Show**

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

```
class Show a where
  show :: a -> String
```

```
data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False

instance Show TrafficLight where
   show Red = "Red light"
   show Yellow = "Yellow light"
   show Green = "Green light"
```