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

Práctica 6: Módulos y Polimorfismo en Haskell (I)

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.

1. Módulos: Importación de módulos

Un programa en Haskell consiste básicamente en una colección de módulos. Como se ha visto previamente, desde un módulo es posible importar otros módulos, utilizando la palabra clave **import**:

```
module ModuleNameX where
  import ModuleNameY
  …
```

que debe escribirse antes de definir cualquier función, por lo que usualmente se pone al principio del propio módulo.

Para poder importar un módulo, es necesario que su nombre coincida con el nombre del fichero que lo contenga, cuando el módulo importado y el que realiza la importación se encuentren en el mismo directorio.

En cambio, si se desea importar cierto módulo que se encuentra en el fichero de nombre "**EjemImport.hs",** situado en el directorio **A/B/C**, relativo al módulo en que se desee hacer la importación, deberíamos escribir:

```
import A.B.C.EjemImport
```

1.1 Lista de exportación

Junto al nombre del módulo, puede aparecer una lista de los elementos del mismo que se deseen exportar (funciones, tipos y constructores) para que puedan ser utilizados por otros módulos, siguiendo la sintaxis:

```
module Nombre (Elem1,Elem2,... ) where
```

Si se omite la lista de exportación, entonces se exporta todo lo definido. Veamos ahora un ejemplo donde se exportan dos funciones definidas dentro del módulo:

```
module Geometry2D (areaSquare, perimeterSquare) where
  areaRectangle :: Float -> Float -> Float
  areaRectangle base height = base * height
  perimeterRectangle :: Float -> Float -> Float
  perimeterRectangle base height = 2 * (base + height)
  areaSquare :: Float -> Float
  areaSquare side = areaRectangle side side
  perimeterSquare :: Float -> Float
  perimeterSquare side = perimeterRectangle side side
```

1.1 Lista de exportación: Compilando con runghc

Si se prueba, a continuación, ejecutar el programa siguiente (escrito en un fichero **Test.hs**):

```
import Geometry2D
main = do putStrLn ("The area is " ++ show (areaRectangle 2 3))
```

Se observa que un programa define una función **main**. Para compilar y ejecutar este programa, en lugar de invocar **main** usando el intérprete, **GHCi**, se puede escribir lo siguiente en la línea de comandos:

```
bash$ runghc Test.hs
```

como se ve, se produce un error:

```
Test.hs:2:55: Not in scope: ‘areaRectangle’
```

Si se modifica, a continuación, la definición de la función main por la siguiente (nuevo contenido del fichero **Test.hs**):

```
import Geometry2D
main = do putStrLn ("The area is " ++ show (areaSquare 2))
```

y se vuelve a probar la ejecución con el comando **runghc**, ya no habrá ningún error.

1.1 Lista de exportación: Compilando con ghc

Además de compilar y ejecutar un programa mediante **runghc**, es posible simplemente compilarlo utilizando **ghc** de la manera siguiente:

```
bash$ ghc --make Test.hs
```

que genera un fichero ejecutable llamado **Test** que se puede ejecutar directamente:

```
bash$ ./Test
The area is 4.0
```

Nota: ghc creará un ejecutable si el código donde está el **main** no es un módulo o bien si este módulo se llama **main**. En caso de tener un módulo con un nombre diferente hay que utilizar la siguiente directiva en el compilador:

```
bash$ ghc -main-is Test --make Test.hs
```

Cuando se quiere utilizar varias instrucciones de salida en una misma función, se pueden agrupar con la notación **do** de la manera que se indica en el programa siguiente (escrito en un fichero **Test2.hs**):

```
import Geometry2D
  main = do
    putStrLn ("Th area is " ++ show (areaSquare 2))
    let other = (areaSquare 5)
    putStrLn ("Another area is " ++ show other)
```

donde la definición de variables dentro del bloque do se realiza mediante **let**.

1.2 Importaciones cualificadas

¿Qué ocurre si dos módulos tienen definiciones con los mismos identificadores? Por ejemplo, supóngase que se tiene el módulo:

```
module NormalizeSpaces where
  normalize :: String -> String
  normalize = unwords . words
```

que utiliza la función **words** para fraccionar una cadena en una lista de palabras (ignorando espacios, tabuladores y enter extras) y la función **unwords** que permite formar de nuevo la cadena a partir de la lista.

Supóngase ahora que hay otro módulo **NormalizeCase** con una función con el mismo nombre:

```
module NormalizeCase where
  import Data.Char (toLower)
  normalize :: String -> String
  normalize = map toLower
```

1.2 Importaciones cualificadas

Importarlos simultáneamente provocaría una colisión de nombres. Para resolver este problema, Haskell permite importar módulos usando la palabra reservada **qualified** que hace que los identificadores definidos por dicho módulo tengan como prefijo el nombre de su módulo, o un alias especificado con **as**:

```
module NormalizeAll where
  import qualified NormalizeSpaces
  import qualified NormalizeCase as NC
  normalizeAll :: String -> String
  normalizeAll =NormalizeSpaces.normalize . NC.normalize
```

Ejercicio 2: Escribir un módulo **Circle.hs** con una función **area** y otro módulo **Triangle.hs** con una función **area**. A continuación, escribir un pequeño programa que importe de manera cualificada la función **area** de cada módulo y que muestre por pantalla el área de un círculo de radio 2 y el área de un triángulo de base 4 y altura 5.

2.1. Polimorfismo paramétrico: Clases de tipos

En la práctica anterior hemos estudiado como crear nuevos tipos algebraicos. Utilizando la palabra reservada **data** podríamos proponer una definición alternativa para el tipo lista:

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

No debe confundirnos el uso de los constructores **Cons** y **Empty**, ya que son otra forma de expresar **:** y **[]** respectivamente. Esta definición de **List** permite definir funciones polimórficas utilizando la variable de tipo **a**.

```
Prelude> Empty
Empty
Prelude> 5 `Cons` Empty
Cons 5 Empty
Prelude> 3 `Cons` (4 `Cons` (5 `Cons` Empty))
Cons 3 (Cons 4 (Cons 5 Empty))
```

En la definición de **List** también se hace referencia a las **clases de tipos**: Show, Read, Eq y Ord. Las clases de tipos pueden considerarse (estableciendo una analogía con Java) una especie de interfaz que define algún tipo de comportamiento.
Si un tipo es miembro de una **clase de tipos** (como en este caso el tipo **List**), significa que ese tipo soporta e implementa el comportamiento que define la clase de tipos.

2.1. Polimorfismo paramétrico: Clases de tipos

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

Show es una **clase de tipos**, que vimos en la sesión anterior, y es utilizada por los tipos que pueden ser representados como cadenas de texto, e implementan las función **show.**

Eq es utilizada por los tipos que soportan comparaciones por igualdad. Los miembros de esta clase implementan las funciones **==** o **/=** en algún lugar de su definición.

Ord es utilizada por los tipos que poseen algún orden entre sus elementos. Los miembros de esta clase implementan las funciones **>**, **>=**, **<** , **<=**.

```
Prelude> 5 == 5
True
Prelude> let a = 5 `Cons` Empty
Prelude> let b = 5 `Cons` Empty
Prelude> a == b
True
```

```
Prelude> [1,2,3,4,5] < [1,2,3,4,5,6]
True
Prelude> let a = 5 `Cons` Empty
Prelude> let b = 6 `Cons` Empty
Prelude> a > b
False
```

2.1. Polimorfismo paramétrico: Restricciones de clase

En Haskell se pueden imponer restricciones a las variables de tipo de forma similar a como hacíamos en Java, mediante las **restricciones de clase**.
Para ejemplificar las restricciones de clase podríamos ahora ver los tipos de algunas funciones:

Vemos que la función **(==)** es una función genérica:

```
Prelude> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

Y tambien vemos algo nuevo, el símbolo **=>**. Cualquier cosa antes del símbolo **=>** es una restricción de clase.
La restricción "**(Eq a) =>**" en la definición anterior hace que ésta se lea:

"para todo tipo **a** que sea una **instancia de** la clase de tipos **Eq**"

Lo mismo es aplicable a la función **(>)**, pero ahora restringida a los tipos que tienen un orden (**Ord**)**:**

```
Prelude> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

La función **elem** permite saber si existe el elemento indicado dentro de la lista.

```
Prelude> :t elem
elem :: (Eq a) => a -> [a] -> Bool
```

2.1. Polimorfismo paramétrico: Tipo Queue

El siguiente ejemplo muestra un módulo donde se define una estructura de datos de tipo cola o **Queue** con las 5 funciones que ya vimos en las prácticas de Java: **enqueue, dequeue, first, isEmpty** y **size**.

```
module Queue (Queue, empty, enqueue, dequeue, first, isEmpty, size) where
  data Queue a = EmptyQueue | Item a (Queue a)
  empty = EmptyQueue
  enqueue x EmptyQueue = Item x EmptyQueue
  enqueue x (Item a q) = Item a (enqueue x q)
  dequeue (Item _ q) = q
  first (Item a _) = a
  isEmpty EmptyQueue = True
  isEmpty _ = False
  size EmptyQueue = 0
  size (Item _ q) = 1 + size q
```

Obsérvese que los módulos que importen **Queue** no podrán utilizar los constructores de los valores del tipo **Queue** (que son **Item** y **EmptyQueue**), puesto que no son visibles (no han sido exportados). En su lugar, se han de crear colas mediante las funciones **empty, enqueue** y **dequeue.**

2.1. Polimorfismo paramétrico: Tipo Queue

Compruébese qué pasa al intentar utilizar uno de los constructores. Para ello, escribir el fichero **TestQueue.hs** como sigue:

```
import Queue
main = do
   putStrLn show (isEmpty (EmptyQueue))
```

e intentar compilarlo, con el consiguiente error por usar **EmptyQueue**:

```
bash$ ghc --make TestQueue.hs
[1 of 2] Compiling Queue ( Queue.hs, Queue.o )
[2 of 2] Compiling Main ( TestQueue.hs, TestQueue.o )
TestQueue.hs:3:28: Not in scope: data constructor ‘EmptyQueue’ 4
```

Sin embargo, este otro ejemplo (fichero **TestQueue2.hs**):

```
import Queue
 main = do
   putStrLn (show (first (enqueue 5 empty)))
```

funciona sin problemas:

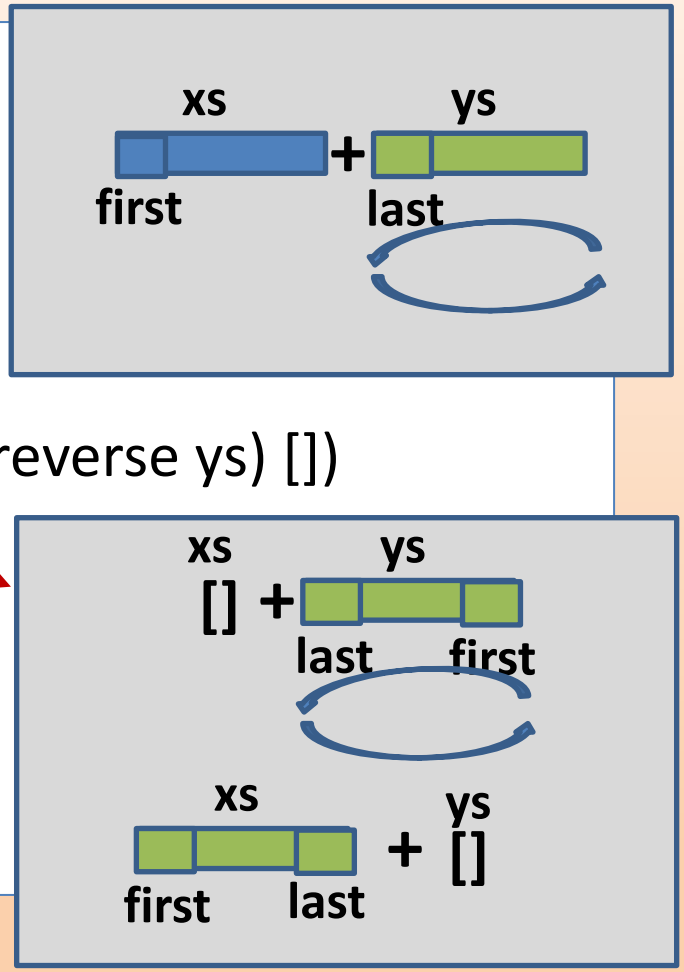
```
bash$ runghc TestQueue2.hs
5
```

Es decir, se pueden ocultar los detalles de la estructura de datos y la definición de las funciones. Esto permite cambiar la implementación sin afectar a quienes hagan uso de **Queue**.

2.1. Polimorfismo paramétrico: Tipo Queue

Así, por ejemplo, también se podría definir **Queue2L** con dos listas:

```
module Queue2L where
  data Queue a = Queue [a] [a]
  empty = Queue [] []
  enqueue y (Queue xs ys) = Queue xs (y:ys)
  dequeue (Queue (x:xs) ys) = Queue xs ys
  dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
  first (Queue (x:xs) ys) = x
  first (Queue [] ys) = head (reverse ys)
  isEmpty (Queue [] []) = True
  isEmpty _ = False
  size (Queue a b) = length a + length b
```



Y los módulos que utilizan el tipo **Queue** seguirían funcionando igual importando ahora el módulo **Queue2L** en vez del módulo **Queue**. En este caso, el tipo de datos algebraico utilizado tiene un solo constructor.

2.1. Polimorfismo paramétrico: Tipo Queue

Supóngase ahora que interesa mostrar una cola (es decir, mostrarla mediante una cadena). Para ello, la forma estándar en Haskell consiste en hacer que la cola sea una instancia de la clase de tipos **Show**, lo cual garantizaría que haya una función de tipo:

```
show :: (Queue a) -> String
```

aunque, si se deseara también mostrar el contenido de la cola sería necesario que el tipo de **a** fuese también de tipo **Show**:

```
show :: (Show a) => (Queue a) -> String
```

Hacer que **Queue** sea instancia de **Show** puede lograrse de manera muy sencilla añadiendo "deriving **Show**" en la declaración del tipo **Queue** lo que proporciona un comportamiento por defecto para las funciones asociadas.

En este caso el comportamiento de la función **show** sería como muestra este ejemplo (fichero **TestQueue3.hs**):

```
import Queue - - o Queue2L
main = do
  putStrLn (show (enqueue 7 (enqueue 5 empty)))
```

que da este resultado (funciona porque **Int** es de la clase de tipos **Show**):

```
bash$ runghc TestQueue3.hs
Item 5 (Item 7 EmptyQueue)
```

Crear instancias de una clase de tipos (**Instance**)

Hay una forma más general de indicar que un tipo es una instancia de una clase de tipos. Se puede definir la función **show** para **Queue**, para ello hay que **añadir lo siguiente al final del módulo Queue**:

```
module Queue (.....) where
 ...
 instance (Show a) => Show (Queue a) where
   show EmptyQueue = " <- "
   show (Item x y) = " <- " ++ (show x) ++ (show y)
```

La función **show** se sobreescribe para el tipo **Queue**. Ahora no debemos hacer "deriving **Show**" en la definición del tipo **Queue** (dará error).

Podemos probar el método **show** con el siguiente ejemplo (fichero **TestQueue4.hs**):

```
import Queue
main = do
  putStrLn (show (dequeue (enqueue 1 empty)))
  putStrLn (show (enqueue 10 (enqueue 5 empty)))
```

que genera la siguiente salida:

```
bash$ runghc TestQueue4.hs
<-
<- 5 <- 10 <-
```

Ejercicios

Ejercicio 3: Considerando la segunda definición del tipo **Queue a** (la que usa el constructor **Queue** [a] [a], con dos listas), definir la función **show** para dicho tipo, de modo que funcione para tipos de **a** que sean de la clase de tipos **Show**. Aunque se podría realizar fácilmente utilizando "deriving **Show**" , se tiene que resolver utilizando **instance**.

Ejercicio 4: Considerando la primera definición del tipo **Queue a** (la que usa los constructores **Item** y **EmptyQueue**), definir la función operador **==** para dicho tipo, de modo que funcione para tipos de **a** que sean de la clase de tipos **Eq**. Aunque se podría realizar fácilmente utilizando "deriving **Eq**" , se tiene que resolver utilizando **instance**.

Puedes utilizar como referencia, la implementación del operador **==** para el tipo lista estándar de Haskell:

```
(==) :: [a] -> [a] -> Bool
[] == [] = True
[] == (x:xs) = False
(x:xs) == [] = False
(x:xs) == (y:ys) = x==y && xs==ys
```

Ejercicios

Ejercicio 5: Considerando de nuevo la primera definición del tipo **Queue a**, definir las funciones **toList** y **fromList** que convierten un valor del tipo **Queue a** en una lista de tipo [**a**] con los elementos de la cola y viceversa. Para ello, se ha de importar el módulo **Queue** y utilizar las funciones que éste exporta (sin recurrir a los constructores **Item** y **EmptyQueue**).