Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2016-17

Parte II Programación Funcional

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Práctica 5: Módulos y Polimorfismo en Haskell

Índice

1. Objetivo de la Práctica

En esta práctica se presenta el uso de los módulos en Haskell y se muestran algunos aspectos básicos del polimorfismo en Haskell. Se han previsto 2 sesiones para resolver los ejercicios planteados en la misma.

2. Módulos

Un programa en Haskell consiste básicamente en una colección de módulos. Un módulo de Haskell puede contener definiciones de funciones, de tipos de datos y de clases de tipos.

Como se ha visto previamente, desde un módulo es posible importar otros módulos, para lo que se utiliza la sintaxis:

```
import ModuleName
```

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

Como se recordará, el nombre de los módulos es alfanumérico y debe empezar en mayúscula. Adicionalmente, el contenido de un módulo empieza, además, con la palabra reservada `module`.

2.1. Importación de módulos

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.

Si el módulo que se desee importar no se encuentra en el mismo directorio que aquel desde el que se realice la importación, entonces es necesario nombrar el módulo a ser importado prefijando su nombre con los de la secuencia (*path*) de directorios para llegar hasta el mismo.

Por ejemplo, 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, entonces el módulo a ser importado debe tener necesariamente por nombre: `A.B.C.EjemImport`.

2.2. Lista de exportación

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

```
module Nombre ( lista de lo que se exporta ) where
```

Si se omite la lista de exportación, entonces se exporta todo lo definido (como se ha hecho hasta el momento). Obviamente, es muy útil poder seleccionar lo que se exporta para poder ofrecer al exterior únicamente un interfaz, ocultando detalles internos no relevantes. Por ejemplo, si se escribe el siguiente módulo en el fichero `Geometry2D.hs`:

```
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
```

Y se prueba, a continuación, ejecutar el programa siguiente (escrito en unfichero `test.hs`):

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

Se observa que un programa define una función `main`. Para ejecutar este programa, en lugar de utilizar el intérprete, GHCi, se debe 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:

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

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

Como se ha observado en este ejemplo, la función `putStrLn` muestra una cadena por salida estándar. Existe otra función llamada `putStr` que es similar a la anterior con la diferencia de que no añade un cambio de línea. 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
```

Cuando se quiere utilizar varias instrucciones de salida en una misma función, se pueden agrupar con la notación `do` de la manera siguiente:

```
import Geometry2D
main = do
  putStrLn ("The 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`.

2.3. 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) -- import only function toLower
  normalize :: String -> String
  normalize = map toLower
```

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:

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

Ejercicio 1 *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.*

3. Polimorfismo en Haskell

3.1. Polimorfismo paramétrico

En prácticas anteriores se han utilizado las listas, cuyo tipo es `[a]`, que es un tipo algebraico (con los constructores `[]` y `:`) y que, además, es polimórfico, puesto que en la expresión `[a]` aparece una variable de tipo: `a`.

Una función es genérica si su tipo contiene variables de tipo. Por ejemplo, la función que calcula la longitud de una lista, que ya es conocida, tiene una implementación que la define para todos los posibles tipos de `a` (esta clase de polimorfismo se conoce como polimorfismo paramétrico):

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Por otra parte, a veces la definición universal para todos los tipos de `a` es demasiado amplia. Por ejemplo, la función de comparación de listas `==` requiere que, a su vez, los valores contenidos en ellas también se puedan comparar (debido a la expresión `x==y` de la siguiente definición que, como se verá, requiere añadir una restricción):

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

Para poder indicar la restricción de que el tipo `a` debe admitir la comparación, Haskell utiliza las *clases de tipos*.

Nota: A pesar del uso de la palabra *clase* en esta denominación, no hay que confundir las clases de tipos de Haskell con el concepto de clase de la programación orientada a objetos. Los tipos no son objetos. Las clases de tipos agrupan ciertos tipos de operaciones, de modo que si un tipo es una instancia de una clase de tipos, hay garantía de que tiene definidas esas operaciones. Esto es más parecido a los interfaces de Java que a sus clases. Un tipo puede ser instancia de varias clases de tipos.

El sistema de clases de tipos de Haskell permite utilizar la parametrización para definir funciones sobrecargadas, imponiendo pertenecer a una clase a los tipos sobre los que se aplica la función. Por ejemplo, la clase `Eq` representa los tipos que tienen definidas las funciones `==` y `/=`. El hecho de que `a` sea de la clase de tipos `Eq` se denota `Eq a`, y se pone como una restricción "`(Eq a) =>`", a la hora de definir el tipo de la función `(==)`, como sigue:

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

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`, la función `(==)` tiene tipo `[a] -> [a] -> Bool`". Es decir, que un tipo sea una instancia de una clase de tipos es garantía de que especifica las operaciones que la clase de tipos indica.

A lo largo de esta práctica se verán ejemplos y se realizarán ejercicios con tipos algebraicos y con clases de tipos. Se verán tanto ejemplos de polimorfismo paramétrico como de polimorfismo *ad hoc* (también conocido como *sobrecarga*). Para más información, se pueden consultar diversos materiales, entre ellos la siguiente página:

`http://www.haskell.org/tutorial/classes.html`

En la definición de las clases de tipos de Haskell no existe la distinción de control de acceso a los métodos que aparecen en Java (`public`, `private`, etc.). En lugar de ello, se utiliza el sistema de módulos que, como se vio en la sección anterior, permite definir listas de elementos a exportar y puede servir para ocultar los detalles de implementación.

El siguiente ejemplo muestra un módulo donde se define una estructura de datos de tipo pila o `Stack` con una serie de funciones para crear una pila vacía (`empty`), añadir y eliminar elementos de la pila (`push` y `pop`), consultar el tope de la pila (`top`) y determinar si la pila está vacía (`isEmpty`):

```
module Stack (Stack, empty, push, pop, top, isEmpty) where
data Stack a       = EmptyStack | Stk a (Stack a)
push x s           = Stk x s
top (Stk x s)      = x
pop (Stk _ s)      = s
empty              = EmptyStack
isEmpty EmptyStack = True
isEmpty (Stk x s)  = False
```

Obsérvese que los módulos que importen `Stack` no podrán utilizar los constructores de los valores del tipo `Stack` (que son `EmptyStack` y `Stk`), puesto que no son visibles (no han sido exportados). En su lugar, se han de crear pilas mediante las funciones `empty`, `push` y `pop`. Compruébese qué pasa

al intentar utilizar uno de los constructores. Para ello, escribir el fichero `testStack.hs` como sigue:

```
import Stack
main = putStrLn show(isEmpty (EmptyStack))
```

e intentar compilarlo, con el consiguiente error del uso del constructor `EmptyStack`:

```
bash$ ghc --make testStack.hs
[1 of 2] Compiling Stack            ( Stack.hs, Stack.o )
[2 of 2] Compiling Main             ( testStack.hs, testStack.o )
testStack.hs:3:30: Not in scope: data constructor ‘EmptyStack’
```

Sin embargo, este otro ejemplo (fichero `testStack2.hs`):

```
import Stack
main = putStrLn (show (top (push 5 empty)))
```

funciona sin problemas:

```
bash$ runghc testStack2.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 `Stack`. Por ejemplo, se puede redefinir la pila utilizando una lista:

```
module Stack (Stack, empty, push, pop, top, isEmpty) where
data Stack a          = Stk [a]
empty                 = Stk []
push   x (Stk xs)     = Stk (x:xs)
pop      (Stk (x:xs)) = Stk xs
top      (Stk (x:xs)) = x
isEmpty  (Stk xs)     = null xs
```

Los módulos que utilizan la pila seguirían funcionando igual. En este caso, el tipo de datos algebraico utilizado tiene un solo constructor: cuando se quiera crear un tipo que es básicamente igual que otro (una lista) pero no es un sinónimo (puesto que las funciones no se definen para una lista), es preferible utilizar `newtype` en lugar de `data`, pero no se va a profundizar en el uso de `newtype` en esta práctica.

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

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

aunque, seguramente, no se querrá mostrar solo una cadena de tipo `"una pila"`, sino que se deseará mostrar el contenido de la pila y, para ello, sería necesario que el tipo `a` fuese también de tipo `Show`:

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

Hacer que `Stack` sea de tipo `Show` puede lograrse de manera muy sencilla: basta con añadir `deriving Show` en la declaración del tipo `Stack` (en este punto, se vuelve a utilizar la implementación inicial):

```
module Stack (Stack, empty, push, pop, top, isEmpty) where
data Stack a = EmptyStack | Stk a (Stack a) deriving Show
...
```

El uso de `deriving` está limitado a un conjunto reducido de clases de tipos estándar (`Eq`, `Show`, `Ord`, `Enum`, `Bounded` y `Read`) y proporciona un comportamiento por defecto para las funciones asociadas. En el caso de tipos algebraicos, sería como muestra este ejemplo (fichero `testStack3.hs`):

```
import Stack
main = putStrLn (show (push 7 (push 5 empty)))
```

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

```
bash$ runghc testStack3.hs
Stk 7 (Stk 5 EmptyStack)
```

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 `Stack`, para ello hay que añadir lo siguiente al final del módulo `Stack`:

```
instance (Show a) => Show (Stack a) where
   show EmptyStack = "|"
   show (Stk x y) = (show x) ++ " <- " ++ (show y)
```

Nota: Obsérvese que, en la definición de la función, aparecen 2 llamadas a `show`, pero que la primera usa la definición de `show` para el tipo `a`, mientras que la segunda es una llamada *recursiva* a la propia función.

Obsérvese también que el carácter "`|`" indica el fondo de la pila como muestra el siguiente ejemplo (fichero `testStack4.hs`):

```
import Stack
main = do
  putStrLn (show (pop (push 1 empty)))
  putStrLn (show (push 10 (push 5 empty)))
```

que genera la siguiente salida:

```
|
10 <- 5 <- |
```

Ejercicio 2 *Definir la función operador* `==` *para el tipo* `Stack a` *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`*.*

Ejercicio 3 *Definir las funciones* `toList` *y* `fromList` *que convierten un valor del tipo* `Stack a` *en una lista de tipo* `[a]` *con los elementos de la pila y viceversa. Para ello, se ha de importar el módulo* `Stack` *y utilizar las funciones que éste exporta (sin recurrir a los constructores* `EmptyStack` *y* `Stk`*).*

3.2. Polimorfismo ad hoc o sobrecarga

Para definir una función cuyo comportamiento dependa del tipo de valor recibido no hace falta necesariamente recurrir a clases de tipos. El siguiente ejemplo muestra cómo se puede definir un tipo de figura geométrica `Shape` que define 2 tipos de figura de modo que el cálculo del área se define según ese tipo:

```
type Height = Float
type Width  = Float
type Radius = Float
data Shape  = Rectangle Height Width |
              Circle Radius
              deriving (Eq, Show)
area :: Shape -> Float
area (Rectangle h w) = h * w
area (Circle r) = pi * r**2
```

Un problema de esta forma de trabajar 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, por ejemplo `Rectangle` y `Circle`. Obsérvese que después se podrán definir términos de los tipos de datos `Rectangle` y `Circle`, por lo que se tendrá una clase, dos instancias y sucesivos términos de esas instancias. La definición usando clases es la siguiente:

```
type Height = Float
type Width  = Float
```

```
type Radius = Float
data Rectangle = Rectangle Height Width
data Circle = Circle Radius

class Shape a where
  area :: a -> Float

instance Shape Rectangle where
  area (Rectangle h w) = h * w

instance Shape Circle where
  area (Circle r) = pi * r**2

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 `Rectangle` 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 función `area` tiene polimorfismo adhoc.

Ejercicio 4 *Modificar la clase de tipos* `Shape` *para que tenga también una función* `perimeter` *que devuelva el perímetro de una figura. Para ello, modifíquense adecuadamente las instancias de las clases* `Rectangle` *y* `Circle`*.*

Ejercicio 5 *Añadir la misma función* `perimeter` *del ejercicio anterior a la definición de figuras basada en tipos algebraicos vista al inicio de este apartado. Es decir, a la definición que incluye*

```
data Shape  = Rectangle Height Width |
              Circle Radius
              deriving (Eq, Show)
```

Ejercicio 6 *Definir una función* `surfacePrism` *que calcule la superficie de un prisma.*

Ejercicio 7 *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
```

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

4. Evaluación

La asistencia a las sesiones de prácticas es obligatoria para aprobar la asignatura. La evaluación de esta segunda parte de prácticas se realizará mediante un examen individual en el laboratorio.