Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2020-21

Parte II Programación Funcional

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Práctica 6 - Material de lectura previa

Módulos y Polimorfismo en Haskell

Índice

1. 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`.

1.1. 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`.

1.2. 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 `Sphere.hs`:

```
module Sphere (area, volume) where

  -- area de una esfera
  area :: Float -> Float
  area radius =  4 * pi * radius**2

  -- volumen de una esfera
  volume :: Float -> Float
  volume radius = (4/3) * pi * radius**3

  -- area de un huso esferico
  areaHuso :: Float -> Float -> Float
  areaHuso radius angle = (area radius) * angle / 360

  -- volumen de la cuña esferica
  volumeCunya :: Float -> Float -> Float
  volumeCunya radius angle = (volume radius) * angle / 360
```

Los programas que importen este módulo (mediante `import Sphere`) podrán usar las funciones exportadas (`area`, `volume`), pero no las otras funciones declaradas en ese módulo (`areaHuso`, `volumeCunya`).

1.3. 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
```

2. Polimorfismo en Haskell

2.1. 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.