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

Parte II Programación Funcional

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Práctica 6: Módulos y Polimorfismo en Haskell

Índice

1. Módulos **2**
1.1. Lista de exportación 2
1.2. Importaciones cualificadas 3

2. Polimorfismo en Haskell **4**
2.1. Polimorfismo paramétrico 4
2.2. Polimorfismo ad hoc o sobrecarga 7

1. Módulos

1.1. Lista de exportación

Considérese el siguiente módulo, escrito 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
```

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 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:3:38: error:
   Variable not in scope: areaRectangle :: Integer -> Integer -> a0
```

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.

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
```

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 ("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`.

Ejercicio 1 *Escribir los 3 programas en los ficheros* `Geometry2D.hs`, `Test.hs` *y* `Test2.hs`*, y ejecutarlos mediante los comandos* `ghc`, `runghc`*, etc., tal como se ha mostrado en las anteriores explicaciones.*

1.2. Importaciones cualificadas

Recuérdese ¿qué ocurre si dos módulos tienen definiciones con los mismos identificadores? Importarlos simultáneamente provocaría una colisión de nombres.

La solución a este problema no es cambiar los identificadores en los módulos importados, pues, además, el usuario de esos módulos podría no tener permisos para modificarlos. La solución, correcta, que proporciona Haskell consiste en importar esos módulos usando la palabra reservada *qualified*. De este modo, los identificadores definidos en cada módulo importado tendrán como prefijo el nombre de su módulo.

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. Polimorfismo en Haskell

2.1. Polimorfismo paramétrico

El siguiente ejemplo muestra un módulo donde se define una estructura de datos de tipo cola o `Queue` con una serie de funciones para crear una cola vacía (`empty`), añadir (`enqueue`) y eliminar (`dequeue`) elementos de la cola, consultar el primer (`first`) elemento de la cola, determinar si la cola está vacía (`isEmpty`), y consultar el número (`size`) de elementos en la cola:

```
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`. 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:3:29: error:
    Data constructor not in scope: EmptyQueue :: Queue a0
```

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`. Así, por ejemplo, también se podría definir `Queue` con dos listas:

```
module Queue 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 `Queue` seguirían funcionando igual. En este caso, el tipo de datos algebraico utilizado tiene un solo constructor.

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

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

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

```
module Queue (Queue, empty, enqueue, dequeue, first, isEmpty, size) where
  data Queue a = EmptyQueue | Item a (Queue 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 `TestQueue3.hs`):

```
import Queue
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)
```

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`:

```
instance (Show a) => Show (Queue a) where
   show EmptyQueue = " <- "
   show (Item 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 que dentro de `instance` no aparece la signatura de la función `show`. De hecho, incluirla daría un error.

Obsérvese también que la cadena "`<-` " indica la cola sin elementos, como muestra 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:

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

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`*.*

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`*).*

2.2. 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 perímetro se define según ese tipo:

```
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
```

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`*).*

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 `Pentagon` y `Circle`. Obsérvese que después se podrán definir términos de los tipos de datos `Pentagon` 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 Side    = Float
type Apothem = Float
type Radius  = Float

data Pentagon = Pentagon Side Apothem
data Circle   = Circle Radius

class Shape a where
   perimeter :: a -> Float

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

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

Ejercicio 7 *Modificar esta clase de tipos* `Shape` *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 función `area` tiene polimorfismo adhoc.

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
```

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