Practical Work of Languages, Technologies, and Paradigms of Programming 2018-19 Part II Functional Programming

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Practice 6: Modules and Polymorphism in Haskell

Contents

1 Modules

1.1 Exportation list

Consider the following module, written in the file `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
```

If you try to execute the following program (written in the file `Test.hs`):

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

you can observe that a program defines a function called `main`. In order to execute this program, instead of using the GHCi interpreter, you have to write the following command:

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

However, you can see that it reports an error:

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

If you modify the definition of the `main` function by the following one (new content of the file `Test.hs`):

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

and try it again with RunGHC we can observe that it now works without problems because `areaSquare` is in the exportation list.

As you have probably observed from this example, the `putStrLn` function shows an string by standard output. There is a related function `putStr` which is similar to the previous one (the last one does not include a newline). Besides compiling and executing the program with `runghc`, it is possible to simply compile it by using `ghc` as follows:

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

which generates an executable file `test` that can be executed from the console:

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

Note: `ghc` will only make an executable if the file containing the `main` function is not a module or if it is a module called `Main`. If there is a module and it is not called `Main` the following compiler option is required:

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

It is possible to group several output instructions in the same function by means of the `do` notation as follows (written in the file `Test2.hs`):

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

where the definition of variables inside the `do` block can be done by using `let`.

Exercise 1 *Write these programs in the files* `Geometry2D.hs`*,* `Test.hs` *and* `Test2.hs`*, and execute them by using the commands* `ghc`*,* `runghc`*, etc., as is previously explained.*

1.2 Qualified Import

Let us remember what happens when two modules have definitions with the very same identifiers. Importing them simultaneously would lead to a name clash.

The solution to this problem is not to change the identifiers in the imported modules since, indeed, the user of these modules would not have permission to do that. The right solution provided in Haskell consists in importing these modules using the reserved word *qualified.* In this way, the identifiers defined in each module would have as prefix the name of its module.

Exercise 2 *Write a module* `Circle.hs` *with a function* `area` *and another module* `Triangle.hs` *with a function* `area`*. Write a short program that imports both functions* `area` *in a qualified way and that prints the area of a circle with radius* 2 *and the area of the triangle with base* 4 *and height* 5*.*

2 Polymorphism in Haskell

2.1 Parametric Polymorphism

The following example shows a module that defines a data structure `Stack` with several functions for creating an empty stack (`empty`), adding or removing elements in the stack (`push` and `pop`), obtain the top element of the stack (`top`) and determine whether or not the stack is empty (`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
```

Let us remark that the modules that import `Stack` cannot use the constructor symbols of the type `Stack` (that is, `EmptyStack` and `Stk`), since they are not visible (not included in the exportation list). Instead, we have to create stacks by using the functions `empty`, `push`, and `pop`. Let us see what happens when we try to use the constructor symbols of `Stack`. Type the following text in the file `TestStack.hs`:

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

if we try to compile it, we get the following error associated to the 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:3:27: Not in scope: data constructor ‘EmptyStack’
```

However, the following alternative example (file `TestStack2.hs`):

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

produces no error:

```
bash$ runghc TestStack2.hs
5
```

That is, we can hide the details of the data structure and the definition of the functions. This allows us to change the implementation without disturbing those modules using the module `Stack`. For example, we can redefine `Stack` using a list:

```
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
```

The modules that use the `Stack` will work as usual. In this case, the algebraic data type that has been used has one constructor: when we want to use a data type that is based on another data type (for example a list) but it is not a synonym (because the functions are not defined on lists) it is recommended to use `newtype` instead of the previous use of `data`, but we are not going to get into details with `newtype` in this practical session.

Let us now imagine that we are interested in showing a stack (that is, to print a string that represents the stack). The standard way in Haskell is to make `Stack` an instance of the class `Show`, which guarantees that there is a function:

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

although, unless we want to show a simple string (e.g. `"a stack"`), we would like to show each element of the stack and, in this case, it would be necessary that the type `a` is also an instance of `Show`:

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

It is very easy to make `Stack` an instance of `Show`: it suffices to add the sentence `deriving Show` to the type declaration of `Stack` (for this we go back to the initial implementation):

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

The use of `deriving` is limited to standard classes (`Eq`, `Show`, `Ord`, `Enum`, `Bounded` and `Read`) and provides a default behavior for each algebraic data type. For instance, in the following algebraic data type (file `TestStack3.hs`):

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

the execution returns the following (it works because `Int` is an instance of the class `Show`):

```
bash$ runghc TestStack3.hs
Stk 7 (Stk 5 EmptyStack)
```

The most general form of specifying that a type is an instance of a class is shown as follows using the type `Stack` as an example. We have to add the following at the end of the definition of the module `Stack`:

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

Note: Observe that, in the definition of the function, there are 2 calls to `show` but the former uses the type definition of `show`, whereas the second call is a *recursive* call to the function itself.

Let us also remark that the signature of the `show` function should not be included inside the block defined by `where`. This would produce a compiler error.

Note also that the character "`|`" is used to indicate the bottom of the stack. You can execute the following example (file `TestStack4.hs`) to check what it does:

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

It generates the following output:

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

Exercise 3 *Considering the first definition of* `Stack a` *(the one using the constructors* `EmptyStack` *and* `Stk`*), define a function* `==` *for the type* `Stack a` *that works for every type* `a` *that is an instance of the clase* `Eq`*. An easy way would be by using* `deriving Eq` *but we want you to use* `instance`*.*

Exercise 4 *Define the functions* `fromList` *and* `toList` *that convert a value of type* `Stack a` *into a list of type* `[a]` *with the elements of the stack and vice versa. You must import the module* `Stack` *and use its exported functions (without using the constructor symbols).*

2.2 Ad-hoc (or overloaded) polymorphism

To define a function whose behavior depends on the given type it is not necessary to use classes. The following example shows how to define a type `Shape` providing two shapes in such a way that the area is calculated depending on the concrete shape:

```
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
```

The problem of defining a type with a constructor symbol for each shape is that it is not possible to dynamically add more constructor symbols to the type `Shape`.

The solution in Haskell is to use a *class* `Shape` and to declare as many instances of this class as shapes we want, for example `Rectangle` and `Circle`. Realize that we could later define terms of type `Rectangle` and `Circle`, so we are considering one class, two type instances and successives terms of these instances. The definition of shapes using classes is as follows:

```
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
```

The `volumePrism` function is able to use elements of the class `Shape a`, concretely terms of the types `Rectangle` and `Circle` (which are instances of `Shape a`) and is able to use the function `area` as well. It will use one of the two functions `area` depending on the actual type. We say that the function `area` has ad-hoc polymorphism.

Exercise 5 *Modify the type class* `Shape` *in order to include a function* `perimeter` *which returns the perimeter of a figure. To this end, modify the instances of the* `Rectangle` *and* `Circle` *classes.*

Exercise 6 *Include the same function* `perimeter` *to the previous exercise with the definition for shapes based on algebraic type. That is, the definition including*

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

Exercise 7 *Define a function* `surfacePrism` *that calculates the surface of a prism.*

Exercise 8 *Modify the definition of* `Shape` *based on type classes in order to allow showing and comparing (equality) two values of the class* `Shape`*. To this end, we have to force that types fulfilling* `Shape` *fulfill also the* `Show` *and* `Eq` *type classes. The idea is to replace the line:*

```
class Shape a where
```

by

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

and by including the required code to make it compile again.