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

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Practice 5: Algebraic types and higher order

Contents

1 Lists

Before resolving the exercises proposed in this section, you should have made a comprehensive reading of the corresponding section, with the same title, of the prior-reading material of this practice.

Let us remember some features of Haskell related to lists:

- The function `length` can be used on lists of any base type:

```
> length [1,2,3]

3

> length ['a','b','c','d']

4
```

- The operator `(!!)` can be used to index lists. This operator can also be used with lists of any type:

```
> [1,2,3] !! 2

3

>  ['a','b','c','d'] !! 0

'a'
```

- The `(++)` operator allows us to concatenate two lists of the same type, for an arbitrary type:

```
> [1,2,3] ++ [4,5,6]

[1,2,3,4,5,6]

> ['a','b','c','d'] ++ ['e','f']

"abcdef"
```

Let us se two solved exercises in order to show how to manipulate lists:

Exercise 1 (Solved) *Define a function to calculate the binary value corresponding to a non-negative integer x:*

```
decBin :: Int -> [Int]
```

For instance, the evaluation of the expression:

```
> decBin 4
```

must return the list `[0,0,1]` *(starting from the least significant bit).*

```
module DecBin where
 decBin :: Int -> [Int]
 decBin x = if x < 2 then [x]
            else (x `mod` 2) : decBin (x `div` 2)
```

Exercise 2 (Solved) *Define a function to calculate the decimal value corresponding to a number in binary form (represented as a list of 1's and 0's, starting again from the least significant bit):*

```
binDec :: [Int] -> Int
```

For example, the evaluation of the expression:

```
> binDec [0,1,1]
```

must return the value `6`.

```
module BinDec where
 binDec :: [Int] -> Int
 binDec (x:[]) = x
 binDec (x:y)  = x + binDec y * 2
```

Now, you have to solve the following exercises. Although in previous exercises we have placed each function in a different module, it is now recommended to place all the functions of this section in the same module.

Exercise 3 *Define a function to calculate the list of divisors of a non-negative number n:*

```
divisors :: Int -> [Int]
```

For example, the evaluation of the expression:

```
> divisors 24
```

must return the list `[1,2,3,4,6,8,12,24]`.

Exercise 4 *Define a function to calculate if an integer belongs to a list of integers:*

```
member :: Int -> [Int] -> Bool
```

For example, the evaluation of the expression:

```
> member 1 [1,2,3,4,8,9]
```

must return the value `True`*. And the evaluation of the expression:*

```
> member 0 [1,2,3,4,8,9]
```

must return the value `False`*.*

Exercise 5 *Define a function to determine whether or not a given number is a prime number (its divisors are just 1 and the number itself) and a function to compute the list of the n first prime numbers:*

```
isPrime :: Int -> Bool

primes :: Int -> [Int]
```

For example, the evaluation of the expression:

```
> isPrime 2
```

must return the value `True`*. The evaluation of:*

```
> primes 5
```

must return the list `[1,2,3,5,7]`*. Let us remember that* Haskell *allows us to easily obtain a list with the first n elements of another infinite list (for instance, the infinite list of prime numbers).*

Exercise 6 *Define a function to* select *the even elements from a list of integers:*

```
selectEven :: [Int] -> [Int]
```

For example, the evaluation of the expression:

```
> selectEven [1,2,4,5,8,9,10]
```

must return the list `[2,4,8,10]`*.*

Exercise 7 *Define a function to* select *the elements that occupy the "even positions" of a list of integers (let us remember that positions in a list start by index zero, remember operator* `!!`*):*

```
selectEvenPos :: [Int] -> [Int]
```

For example, the evaluation of the expression:

```
> selectEvenPos [1,2,4,5,8,9,10]
```

must return the list `[1,4,8,10]`.

Exercise 8 *Define a function* `iSort` *to sort a list in ascending order. To this end, you first have to define a function* `ins` *which inserts an element in an ordered list keeping the list ordered. The function* `iSort` *should be based on* `ins` *starting from an empty list (which is trivially sorted).*

```
iSort :: [Int] -> [Int]

ins :: Int -> [Int] -> [Int]
```

For example, the evaluation of the expression:

```
> iSort [4,9,1,3,6,8,7,0]
```

must return the list `[0,1,3,4,6,7,8,9]`. *And the evaluation of the expression:*

```
> ins 5 [0,1,3,4,6,7,8,9]
```

must return the list `[0,1,3,4,5,6,7,8,9]`.

2 The `map` and `filter` functions

Before resolving the exercises proposed in this section, you should have made a comprehensive reading of the corresponding section, with the same title, of the prior-reading material of this practice.

Exercise 9 *Define, using the function* `map`, *a function to duplicate all the elements of a list of integers:*

```
doubleAll :: [Int] -> [Int]
```

For example, the evaluation of the expression:

```
> doubleAll [1,2,4,5]
```

returns the list `[2,4,8,10]`.

Exercise 10 *Write using intensional lists the definitions of the functions* `map` *and* `filter`. *Note: name them as* `map'` *and* `filter'` *to avoid conflicts with the functions predefined in* `Prelude`.

3 Algebraic Data Types

Let us remember once again that you should have made a comprehensive reading of the corresponding section, with the same title, of the prior-reading material of this practice.

3.1 Enumerations and Renamed Types

In order to solve the exercise proposed in this section, we have to define previously the following "synonym types":

```
type Person = String

type Book = String

type Database = [(Person,Book)]
```

The type `Database` defines a database of a library as a list of pairs `(Person,Book)` where `Person` is the name of the person that borrowed the book `Book`. An example is the following database

```
exampleBase :: Database

exampleBase = [("Alicia","El nombre de la rosa"),

               ("Juan","La hija del canibal"),("Pepe","Odesa"),

               ("Alicia","La ciudad de las bestias")]
```

From this database, several functions can be defined: `obtain` for the books borrowed by a person, `borrow` for a person to borrow a book, and `return'` for retuning a book to the library. For example, the function `obtain` can be defined as

```
obtain :: Database -> Person -> [Book]

obtain dBase thisPerson

 = [book | (person,book) <- dBase, person == thisPerson]
```

which represents that the function returns the list of all the books such that there is a pair `(person,book)` in the database and `person` is equal to the person whose books we are searching for. For example, the evaluation of the expression: `obtain exampleBase "Alicia"` returns the list: `["El nombre de la rosa","La ciudad de las bestias"]`

Exercise 11 *Create a module with the previous code and complete the program with definitions for the* `borrow` *and* `return'` *functions:*

```
borrow :: Database -> Book -> Person -> Database

return' :: Database -> (Person,Book) -> Database
```

3.2 Trees

Let us remember the declaration of tree datatypes already described in the prior-reading material. They will be used in the rest of exercises of this practice:

```
data TreeInt = Leaf Int | Branch TreeInt TreeInt
```

- Example:
  `Branch (Leaf 0) (Branch (Leaf 0) (Leaf 1))`
  is a value of type `TreeInt`.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

- Example:

  ```
  numleaves (Leaf x)     = 1

  numleaves (Branch a b) = numleaves a + numleaves b
  ```

  is a function that computes the number of leaves of a tree of type `Tree a`.

```
data BinTreeInt  = Void | Node Int BinTreeInt BinTreeInt
```

- Let us see some examples:

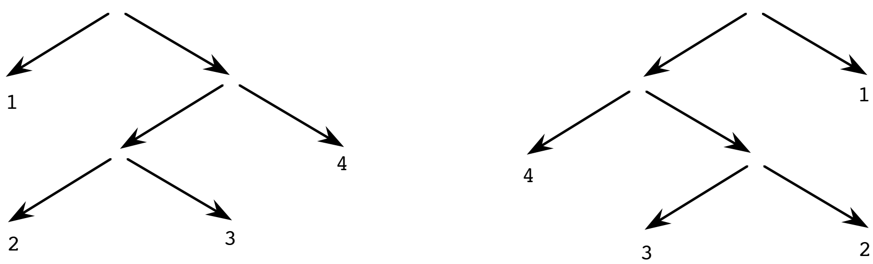
  ```
  treeB1 = Void

  treeB2 = (Node 5 Void Void)

  treeB3 = (Node 5
             (Node 3 (Node 1 Void Void) (Node 4 Void Void))
             (Node 6 Void (Node 8 Void Void)))
  ```

Exercise 12 *Define the following function to obtain the symmetric of a given tree:*

`symmetric :: Tree a -> Tree a` *where the symmetric of a tree (on* *the left) is as illustrated (on the right) in the following figure:*



Tip: If you try `symmetric` with `ghci` you will obtain the following message:

```
> symmetric (Branch (Leaf 5) (Leaf 7))

<interactive>:5:1:

    No instance for (Show (Tree a0))

      arising from a use of ‘print’

    Possible fix: add an instance declaration for (Show (Tree a0))

    In a stmt of an interactive GHCi command: print it
```

the reason is that `ghci` does not know how to "show" the result. This result is written using the function `show` (in a certain way, it is like `toString` method in Java). We will use a very simple mechanism for specifying a default behavior to the `show` function for a given algebraic data type. Simply add `deriving Show` at the end of declaration of the corresponding algebraic data type, as illustrated in the following example:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
```

Exercise 13 *Define the functions*

```
listToTree :: [a] -> Tree a

treeToList :: Tree a -> [a]
```

the first one converts a non-empty list into a tree, and the second one performs the opposite.

Now, solve the following exercises related to the `BinTreeInt` data type. It is also recommended to add `deriving Show` at the end of the declaration of this type in order to be able to show the result of the functions.

Exercise 14 *Define a function*

```
insTree :: Int ->  BinTreeInt -> BinTreeInt
```

to insert an integer value into its corresponding place in an ordered binary tree..

Exercise 15 *Given an unordered integer list, define a function*

```
creaTree :: [Int] ->  BinTreeInt
```

that builds the ordered binary tree associated to it.

Exercise 16 *Define the function*

```
treeElem :: Int ->  BinTreeInt -> Bool
```

that determines "in an efficient way" whether a value belongs to a ordered binary tree or not.

Additional Exercises

The following exercises can be considered an extension since their resolution could exceed the 2 lab sessions. However, their resolution is highly recommended as a preparatory work for the evaluation.

Exercise 17 *Define a function to determine how many times an element is repeated in a list of integers:*

```
repeated :: Int-> [Int] -> Int
```

For example, the evaluation of the expression:

```
> repeated 2 [1,2,3,2,4,2]
```

must return the value `3`.

Exercise 18 *Define a function to* concatenate *lists of lists, which takes a list of lists as the argument and returns the concatenation of all lists:*

```
concat' :: [[a]] -> [a]
```

For example, the evaluation of the expression:

```
> concat' [[1,2],[3,4],[8,9]]
```

must return the list `[1,2,3,4,8,9]`

Exercise 19 *What does the following mysterious expression compute? (where* `sum` *is a predefined function which sums the elements of a list of numbers):*

```
> (sum . map square . filter even) [1..10]
```

Exercise 20 *Write in a file the definition of the* `numleaves` *function presented before as an example of* `Tree a`*. Load it on the* GHCi *interpreter and consult the type of this function according to the interpreter.*

Exercise 21 *Consider the following declaration of a binary tree of integers described before:*

```
data BinTreeInt  = Void | Node Int BinTreeInt BinTreeInt deriving Show
```

Define a function `dupElem` *that returns a tree with the same structure but where all values are duplicated (multiplied by two).*

Let us consider that `dupElem` *is applied to previously declared trees. The evaluation of the expression:*

```
> dupElem treeB1
```

returns `Void`*. The evaluation of the expression:*

```
> dupElem treeB2
```

returns `Node 10 Void Void`*. And the evaluation of the expression:*

```
> dupElem treeB3
```

returns:

```
Node 10
(Node 6 (Node 2 Void Void) (Node 8 Void Void))
(Node 12 Void (Node 16 Void Void)).
```

Exercise 22 *Let us consider the following definition:*

```
data Tree a = Branch a (Tree a) (Tree a) | Void deriving Show
```

Define a function `countProperty` *with the following signature:*

```
countProperty :: (a -> Bool) -> (Tree a) -> Int
```

which returns the number of elements of the tree that satisfy the property. For instance, the evaluation of the expression:

```
> countProperty (>9) (Branch 5 (Branch 12 Void Void) Void)
```

returns `1`*, and the evaluation of the expression:*

```
> countProperty (>0) (Branch 5 (Branch 12 Void Void) Void)
```

returns `2`*.*