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

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Practice 5 – Prior-reading Material

Algebraic types and higher order

Contents

1 Types and Type Inference

In functional programming, the values generated by expression evaluation are organized in *types*. Each type represents a set of values. For example, the (primitive or basic) type `Int` represents the values `0`, `1`, `2`, `-1`, `-2`, etc. The type `Bool` represents the boolean values `True` and `False`. The type `Char` represents all the alphanumeric characters `'a'`, `'b'`, `'A'`, `'B'`, `'0'`, ... In functional programming, as in the majority of programming languages, *a type is associated to each expression.* This can be explicitly expressed by the typing operator '`::`'.

```
42 :: Int

6*7 :: Int

abs (2*(-2)) :: Int
```

As presented in the previous session, we can ask the GHCi interpreter for the type of any well-formed expression using the user command "`:t`":

```
> :t 'a'

'a' :: Char


> :t "Hello"

"Hello" :: [Char]
```

In functional programming, function identifiers are valid expressions due to currying and they have associated a type, as for other expressions. Moreover, it is not necessary for the programmer to provide explicit typing information for function definitions or written expressions, since the interpreter can automatically *infer* them from their definition. You can try, for example, the following commands:

```
> :t show

> :t (+)

> :t (*)

> :t (3 +)

> :t (* 2.0)
```

Note: Check the particular case of a binary function `(*)` that expects two arguments of a numeric type and returns a value of the same numeric type.

```
Prelude> :t (*)
(*) :: (Num a) => a -> a -> a
```

However, the unary function `(* 2.0)` expects just one argument and returns its double, that is, the number resulting from multiplying the argument by the real constant 2.0. The argument is restricted from `Num` to a more restricted type:

```
Prelude> :t (* 2.0)
(* 2.0) :: (Fractional a) => a -> a
```

Let us observe that this function has been obtained as a simple partial application of the multiplication operator `(*)`, which is a binary operator, to the particular case where the second argument is fixed to the real constant 2.0. This is possible because the function is curried, that is, it expects the arguments as a sequence of expressions separated by juxtaposition rather than a sequence of expressions separated by the comma and enclosed in parenthesis.

2 Reduction Strategy

The reduction strategy in Haskell is called *lazy*. This strategy reduces an expression only if the reduction is really necessary to obtain the result. That is, function arguments are reduced as much as it is necessary to apply a reduction step on the outermost function.

Thanks to this strategy, it is possible to use infinite data structures. In programming languages that use the *eager* reduction strategy, as it is done in imperative programming languages and older functional programming languages, infinite data structures are not available.

The function `repeat'` returns an infinite list (similar to the function `repeat` from the `prelude`):

```
repeat' :: a -> [a]
repeat' x = xs where xs = x:xs
```

The call `repeat' 3` returns the infinite list `[3,3,3,3,3,3,3,3,3,....`

Caution: The call `repeat' 3` does not terminate and prints the infinite list with the number 3, having to stop the execution with `^C`.

The infinite list generated by `repeat'` can be used as a (partial) argument to a function that returns a finite result. The following function, for example, takes a finite number of elements from a (possibly infinite) list:

```
take' :: Int -> [a] -> [a]
take' _ [] = []
take' n (x:t)
   | n<=0 = []
   | otherwise = x : take' (n - 1) t
```

For example, the call `take' 3 (repeat' 4)` returns the list `[4,4,4]`.

3 The lists

3.1 The `List` Data Type

In functional programming, it is possible to use structured data types whose values are built from objects of other types. For example, the list type `[a]` can be used to group objects of the *same* type (denoted, in this case, by the type variable `a`, which can be instantiated to any type). In Haskell, lists can be specified by separating its elements by commas and enclosing them in square brackets:

```
    [1,2,3] :: [Int]

    ['a','b','c','d'] :: [Char]

    [cos,log,(1.0 +)] :: [Float -> Float]

              -- Function list from reals to reals:

              -- e.g. cosine, logarithm in base 2,

              -- increment a real number by one.
```

However, the following lists

```
    [1,'a',2]

    ['a',log,3]

    [cos,2,(*)]
```

are invalid (why? what does GHCi answer when you ask for their type?).

The empty list is denoted by `[]`. When it is not empty, we must decompose it using the notation that splits the list into its first element and the rest of the list. For example:

```
1:[2,3]   or   1:2:[3]   or   1:2:3:[]

'a':['b','c','d']  or  'a':'b':['c','d'] or 'a':'b':'c':'d':[]

cos:[log]   or   cos:log:[]
```

Types that include type variables in their definition (as in the List type) are called generic types or *polymorphic types*. It is possible to define functions on polymorphic types. These functions can be used on objects of any type that is an instance of the corresponding polymorphic type. For instance, the (predefined) function `length` obtains the length of a list:

```
> :t length

length :: [a] -> Int
```

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

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

3

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

4

> length [cos,log,sin]

3
```

The operator `(!!)` can be used to index lists. It can be used to extract the element occupying a given position in a list:

```
> :t (!!)

(!!) :: [a] -> Int -> a
```

This operator can be used with lists of any type:

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

3

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

'a'
```

Another very useful function for lists is `(++)`, which concatenates two lists of the same type, no matter their type:

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

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

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

"abcdef"
```

String characters introduced in previous practice (for instance, `"hello"`) are just list of characters writen in a special syntax which omits the single quotation marks. In this way, `"hello"` is just a convenient syntax for the list `['h','e','l','l','o']`, so any generic operation on lists can also be applied to strings.

3.2 Ranges

The basic form has the syntax `[first..last]` so that it generates the list of values between both (inclusive):

```
> [10..15]

[10,11,12,13,14,15]
```

The syntax of Haskell ranges allows the following options:

- `[first..]`
- `[first,second..]`
- `[first..last]`
- `[first,second..last]`

You can guess the behavior from the following examples:

```
[0..] -> 0, 1, 2, 3, 4, ...

[0,10..] -> 0, 10, 20, 30, ...

[0,10..50] -> 0, 10, 20, 30, 40, 50

[10,10..] -> 10, 10, 10, 10, ...

[10,9..1] -> 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

[1,0..] -> 1, 0, -1, -2, -3, ...

[2,0..(-10)] -> 2, 0, -2, -4, -6, -8, -10
```

3.3 Intensional Lists

Haskell provides an alternative notation for lists, the so called *intensional lists*, which is also useful to describe computations requiring `map` and `filter`, as we shall see in the next section. For example:

```
> [x * x | x <- [1..5], odd x]

[1,9,25]
```

The expression can be understood as: *the list of the squares of the odd numbers in the range from 1 to 5.*

Formally, an intensional list is of the form `[e|Q]`, where `e` is an expression and `Q` is a *qualifier*. A qualifier is a (possibly empty) sequence of *generators* and *guards* separated by commas. A generator takes the form `x <- xs`, where `x` is a variable or a sequence of variables, and `xs` is an expression of type list. A guard is a boolean expression. The qualifier `Q` can be empty and, in that case, we just write `[e]`. Some examples are:

```
> [(a,b) | a <- [1..3], b <- [1..2]]

[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]

>

> [(a,b) | a <- [1..2], b <- [1..3]]

[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
```

The last generators can depend on variables introduced by the former ones, such as the following example:

```
> [(i,j) | i <- [1..4], j <- [i+1..4]]

[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

We can freely combine generators and guards:

```
> [(i,j) | i <- [1..4], even i, j <- [i+1..4], odd j]

[(2,3)]
```

4 The `map` and `filter` functions

`map` and `filter` are two predefined functions very useful for list manipulation.

4.1 The `map` function

The `map` function applies a function, given as an argument, to each element of the list. They constitute examples of *high order functions* since they accept *functions* rather than values as their first argument, as can be observed in the following examples:

```
> map square [9,3]
[81,9]
> map (<3) [1,2,3]
[True,True,False]
```

where `square` is the function to calculate the square of a given integer number (this function could be defined, for instance, as `let square = (^2)`). The definition of `map` is:

```
map          :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

There exist some useful algebraic laws for `map`, two basic facts are:

$$\texttt{map}\ id = id$$
$$\texttt{map}\ (f \cdot g) = (\texttt{map}\ f) \cdot (\texttt{map}\ g)$$

where *id* is the identity function and "$\cdot$" is the function composition. The first equation says that when the identity function is applied by `map` to each element in the list, the resulting list is unchanged. The two occurrences of `id` in the first equation have different types: the left one is `id :: a -> a` and the right one is `id :: [a] -> [a]`. The second equation says that applying g first to each element in the list and then applying f to each element in the list gives the same result than applying the combined function $f \cdot g$ to the original list. This allows the replacement of two traversals of a list by just one traversal with the corresponding efficiency gain, so that it is always recommended.

4.2 The `filter` function

The `filter` function takes a boolean function `p` and a list `xs` and returns a sub-list `xs` whose elements satisfy `p`. For example:

```
> filter even [1,2,4,5,32]
[2,4,32]
```

The definition of `filter` is:

```
filter          :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) = if p x then x:filter p xs else filter p xs
```

5 Algebraic Data Types

5.1 Enumerations

In a functional language such as Haskell, the programmer can define new data types with their associated values using the so called *algebraic data types*. For example, the definition:

```
data Color = Red | Green | Blue
```

establishes a new data type called `Color`. This type `Color` contains only three values or data, denoted by the corresponding constructor constants `Red`, `Green` and `Blue`. Recall that, in Haskell, identifiers of data types and constructors always start with an *uppercase letter* whereas variables always start with a *lowercase letter*.

5.2 Renamed Types

It is also possible to rename previously defined types. This is known as "synonym types". For example:

```
type Distance = Float
type Angle = Float
type Position = (Distance,Angle)
type Pairs a = (a,a)
type Coordinates = Pairs Distance
```

Indeed, the type `String` is a renaming, as we already mentioned in a previous session:

```
type String = [Char]
```

and all list operations can be applied, for example order comparisons (lexicographic order):

```
> "hola" < "halo"

False

> "ho" < "hola"

True
```

Often, we would like to output strings without the double quotes and where special characters, such as carriage return, are printed as they really appear. Haskell provides *commands* for printing strings, reading files, etc. For example, using the function `chr` that converts an integer into the character that it represents:

```
> putStr "Sentence with a" ++ [chr 10] ++ "carriage return"

Sentence with a

carriage return
```

Remember the need to import the `Data.Char` module in order to use the `chr` function.

5.3 Trees

Several definitions for the tree data type can be envisaged. We will consider two of them in the following sections.

5.3.1 One type of tree: `TreeInt` and `Tree a`

The definition:

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

establishes a new data type called `TreeInt`, where `TreeInt` is again a type constructor that contains an infinite set of values, recursively defined using the constructors `Leaf` (unary) and `Branch` (binary) which take an integer number and two `TreeInt` trees as arguments, respectively.

Let us consider some examples of values or data from the previous types:

- `[Red,Green,Red]` is a value of type `[Color]`
- `Branch (Leaf 0) (Branch (Leaf 0) (Leaf 1))` is a value of type `TreeInt`.

It is also possible to define generic types. For instance, the type

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

is an algebraic polymorphic type for defining trees of any kind of data, in a similar way to lists, which also admit any kind of data.

It is possible to define functions over types defined by the user. The function `numleaves` is defined as follows:

```
numleaves (Leaf x)     = 1

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

It computes the number of leaves of a trees of type `Tree a`.

5.3.2 Another type of trees: `BinTreeInt`

Let us consider now the following alternative definition of the data type for integer binary trees:

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

in which integer values are stored in the nodes and the leaves are just empty subtrees (denoted by the constructor symbol `Void`). 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)))
```

you can observe that `treeB1` is an empty tree, `treeB2` is a tree contaning just one element, and `treeB3` is a tree where value 5 is placed at the root, values 3, 1, 4 belong to the left branch, and values 6 and 8 are placed at the right branch.

We would say that a binary tree is ordered (in this case it is usually known as *binary search tree*) if the values stored in the left subtree of a node are always lower or equal to the value of the node, whereas the values stored in right tree of the node are always bigger.