Practical Work of Languages, Technologies, and Paradigms of Programming

2018-19
Part II Functional Programming

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

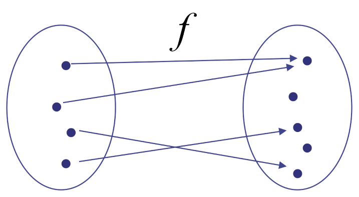
Practice 4 – Prior-reading material

Índice

1. What is Haskell

Haskell is a pure functional programming language. Traditional imperative programming languages (C, C++, Java, C#, etc.) focus our attention in *how* to solve a problem, meaning that the solutions to the problems are expressed as a sequence of tasks which have to be performed to solve the problem. Contrarily, functional programming languages put the emphasis in *what*, that is, in the description of the problem elements as well as the relationships among these elements.

Functions of functional programming languages are based in the mathematical concept of function. Let us remember that mathematical functions are defined between two sets of elements known, respectively, as *domain* and *co-domain*.



In pure functional programming languages, such as Haskell:

- Functions lack side effects. Their execution cannot modify global elements. In fact, they always return the same result for a given input. On the other side, the result of a function only depends on the input arguments (this is known as *referential transparency*). Two functions are equal if and only if they returns the same for the same input arguments.

- The domain and the co-domain of a function may contain another function. Otherwise stated, arguments and results of functions can also be functions. Moreover, an expression can result in a function. For instance:

```
Prelude> let f = (3.5 *)    -- f is "multiply by 3.5"
Prelude> f 5                -- f is applied to 5
17.5
Prelude>
```

- The order functions are defined is not relevant (although the same function can be specified, in some cases, in a series of cases where the order may matter).

- The lack of side effects makes it easier to deal with parallel and concurrent programming.

Besides, Haskell is a strongly typed language. This means that all expressions and functions have a known type at compile time. Any function has a *signature* or profile, the specification of its domain and its co-domain. For instance:

```
example :: String -> Int
```

determines that function `example` is defined from strings (`String`) to integers (`Int`). Additionally, Haskell can infer the types of expressions and, in most cases, the signatures o functions from the explicit and implicit types of the sub-expressions appearing in their definition.

A particular feature of Haskell it the use of *lazy evaluation*, which means that Haskell will delay the actual computation of any expression until the result is really required. For instance, the execution of the following lines of Haskell code:

```
Prelude> let m7 = [x | x <-[1..], x ‘mod‘ 7 == 0]
Prelude> take 10 m7
[7,14,21,28,35,42,49,56,63,70]
Prelude>
```

where the expression `[1..]` represents, in Haskell, the list of all integers from 1, which is obviously an infinite sequence. The first line of code associates to the `m7` identifier the list of all multiples of 7.

The expression `take n ls` returns the first `n` elements from the list `ls`. The fist line makes use of the `let` construct to assign in the interactive environment (the GHCi interpreter) in order to assign to the `m7` identifier an *intensional list* specifying the list of all multiples of 7. Ranges and intensional lists wil be studied into more detail in the next practices.

As can be observed, the computation of the first line has been delayed depending on what is actually required by the second one. In this way, Haskell has only computed the very first 10 elements of an infinite list since the rest are not actually required to compute `take 10`.

2. Simple data types

There exists a collection of data types, functions and operators that can be used in every program. These are predefined elements of the Haskell language and can be found in the `Prelude` module. All predefined functions (except for the arithmetic ones) are prefix although, as we will see, it is possible to use them in infix notation. It is also possible to omit the parenthesis around the arguments of a function call if there is no confusion possible, that means `f (a) (b)` can be written as `f a b`.

1. **The type `Bool`**

   Values of this type represent logical expressions whose result can be true or false. This type only has two constant values: `True` and `False` (written this was), that represent the two possible results.

   Functions and Operators

   The following predefined operators and functions operate with boolean values:

   - `(&&) :: Bool -> Bool -> Bool`. Is the logical conjunction.
   - `(||) :: Bool -> Bool -> Bool`. Is the logical disjunction.
   - `not :: Bool -> Bool`. Is the logical negation.
   - `(==) :: Bool -> Bool -> Bool`. Returns `True` if the first argument is equal to the second one, and `False` if they are different.
   - `(/=) :: Bool -> Bool -> Bool`. Returns `True` if the first argument is not equal to the second one, and `False` if they are equal.

2. **The type `Int`**

   The values of this type are integers with limited[1] precision. The constant values of this type are written using standard notation.

   Functions and Operators

   Some predefined operators and functions for this type are:

   - `(+)`, `(-)`, `(*) :: Int -> Int -> Int`. Are the addition, subtraction and multiplication, respectively.
   - `(^) :: Int -> Int -> Int`. Is the power operator. The exponent must be a natural number.
   - `div`, `mod :: Int -> Int -> Int`. Are the quotient and the rest operator, respectively.
   - `abs :: Int -> Int`. Is the absolute value.
   - `signum :: Int -> Int`. Returns 1, -1 or 0 depending on the sign of the argument.
   - `even`, `odd :: Int -> Bool`. Check whether the argument is even or odd.

[1]We can also use in Haskell the `Integer` type which has non-limited precision and which has the same operators as `Int`. Both types are compatible between them.

- `(==) :: Int -> Int -> Bool`. Returns `True` if the first argument is the same as the second, and `False` if they are different.
- `(/=) :: Int -> Int -> Bool`. Returns `True` if the first argument is the not same as the second, and `False` if they are the same.

3. **The type `Float`**

   Values of this type represent real numbers with a precision limited to 7 or 8 decimal digits[2]. There are two ways to write real values:

   - the usual notation, for example `1.35`, `-1.0`, or `1`.
   - scientific notation, like for example`1.5e3` (that denotes the value `1.5`$\times$ `10`^{3}).

   Functions and Operators

   Some predefined functions and operators are below.

   - `(+)`, `(-)`, `(*)`, `(/) :: Float -> Float -> Float`. Are addition, subtraction, multiplication and division, respectively.
   - `(==) :: Float -> Float -> Bool`. Returns `True` if the first argument is the same as the second, and `False` if they are different.
   - `(/=) :: Float -> Float -> Bool`. Returns `True` if the first argument is the not same as the second, and `False` if they are the same.
   - `sqrt :: Float -> Float` returns the square root.
   - `(^) :: Float -> Int -> Float`. Is the power operator with exponent a natural number.
   - `(**) :: Float -> Float -> Float`. Is the power operator with exponent a real number.
   - `truncate :: Float -> Int`. Gives the integer part of a real.
   - `signumFloat :: Float -> Int`. Returns 1, -1 or 0 depending on the sign of the argument.

4. **The type `Char`**

   A value of type `Char` represents a character (letter, digit, ...). A constant value of this type is written in between simple quotes (for example `'a'`, `'9'`, ...). To use this data type, the module has to include

[2]There is also in Haskell the `Double` type with more precision. However, both types are not compatible between them so that explicit conversion operations are required.

"`import Data.Char`" in the module we are defining or by using the `:module +` command in the current session when using GHCi.

Functions and Operators

Some predefined functions for this type are:

- `ord :: Char -> Int`. Returns the ASCII/Unicode code that corresponds to the character argument.
- `chr :: Int -> Char`. Is the reverse of `ord` function.
- `isUpper, isLower, isDigit, isAlpha :: Char -> Bool`. They check whether the character argument is upper case, lower case, a digit or alphanumerical, respectively.
- `toUpper, toLower :: Char -> Char`. They convert the argument character to upper case or lower case respectively.
- `(==) :: Char -> Char -> Bool`. Returns `True` if the first argument is the same as the second, and `False` if they are different.
- `(/=) :: Char -> Char -> Bool`. Returns `True` if the first argument is the not same as the second, and `False` if they are the same.

3. Modules

A Haskell program is a set definitions of data types, functions and so on. This is organized as a collection of modules. From a syntactic point of view, a modules starts with the keyword `module` followed by the module name (which must start in **uppercase**) and the word `where` which opens a block, as in the following example:

```
module Signum where
  -- definition of signum' function:
  signum' x = if x < 0 then -1 else
              if x == 0 then 0 else 1
```

One of the ways to create comments in Haskell is to use the characters `--` to start a comment until the end of the line, as shown in the previous example.[3] Observe also that in Haskell each `if` must be accompanied by its corresponding `else`.

[3]We can also create comments scoping several lines by using the symbol `{-` to open the comment and `-}` to close it.

Haskell makes use of indentation to mark block endings as in other languages such as Python and unlike others like Java, which makes use of curly braces (`{` and `}`) to this end.

For very simple examples, we can write Haskell code in a file without defining a module. In that case, this is like defining a module called `Main`.

4. Function definitions

A function definition consists in general of:

- A type declaration (of which we have seen already that is optional since GHCi automatically infers types, however it is recommendable to include it in the program).
- A number of equations that define the functionality of the function.

In Haskell we distinguish symbols that are defined through a function definition (like for example `signum'` that we have seen before) and symbols that are constructors. Constructors are symbols that are not defined through a function definition, but are used to construct values of a specific type. The following can be seen as a general schema for a function declaration:

```
f :: Type₁ -> ... -> Typeₙ -> Type_f
f (pattern₁) ... (patternₙ) = expression
```

Each of the expressions $\texttt{pattern}_i$ represents an argument of the function. A pattern can only contain constructors and variables, it cannot contain defined functions. The names of the variables need to be in lower case. Note the difference between `add`$(1, 2, 3)$, syntax that would be used in most languages, and y "`add` 1 2 3", functional syntax (see Section 5).

When there is no ambiguity, parenthesis around the patterns are not needed and can be left out. For example, we can introduce the following function definition for calculating the length of a list:

```
module Length where
  length' [] = 0
  length' (x:t) = 1 + length' t
```

Note: a list may be seen as a sequence of expressions that are connected by the symbol constructors `:` and end with `[]`. For example, the string `"hello"` is actually a list of characters: `['h','e','l','l','o']`, in Haskell this can also be written as: `'h':'e':'l':'l':'o':[]`. Lists are studied into more detail in the next practice.

Once we have loaded this program in the interpreter, we can ask it to evaluate for example the expression `length' ([1,2,3])` that returns the value `3`. In this case we can also write the same expression without parenthesis (i.e. `length' [1,2,3]`), since `length'` has only one argument and there is no ambiguity.

Moreover, the function `length'` admits all types of lists, for example `length'("hola")` returns `4` (there are 4 characters in the list), but `length'(["hola"])` returns `1` (1 string in the list).

It is also possible to define a function using *conditional equations*, the notation is:

$$
\begin{array}{ll}
\texttt{f } \texttt{x}_1\ \texttt{x}_2\ \texttt{...}\ \texttt{x}_n & \\
 & \mid\ condition_1 = exp_1 \\
 & \mid\ condition_2 = exp_2 \\
 & \vdots \\
 & \mid\ condition_m = exp_m
\end{array}
$$

For example, for the power function, we can define:

```
module Power1 where
  power1  :: Int -> Int -> Int
  power1 _ 0 = 1
  power1 n t = n * power1 n (t - 1)
```

where the symbol "_" represents a variable (like `n` or `t`) whose name is irrelevant. A more efficient version using conditional equations is:

```
module Power2 where
  power2  :: Int -> Int -> Int
  power2 _ 0 = 1
  power2 n t
         | even t = power2 (n * n) (div t 2)
         | otherwise =  n * power2 (n * n) (div t 2)
```

where `even` and `div` are predefined functions and the expression `otherwise` always evaluates to `True`.

Note: The order in which the equations appear in a program is important since Haskell looks for the first applicable equation from top to bottom. If it finds one, it will not try the rest. The same happens with conditional equations, where the conditions are evaluated from top to bottom. Again if one evaluates to true, the other conditions will not be evaluated. For example, consider again the function `power2`, if the second argument is a `0`, the first equation will be applied and the second equation will not be tried. If the second argument is not a `0`, the first equation is not applicable and hence the second will be tried. In the conditional equation, the condition `even t` will be tried first. If it evaluates to true, the right part of the equation is returned. If it evaluates to false, the right part of the `otherwise` will be evaluated.

Other expressions that we can use when defining functions in Haskell are:

where We use the expression (equation) `where` when we want to define an expression local to a function[4]. The general schema is:

```
f x₁ x₂ ... xₙ = exp
     where
         definitionFunction₁
         ...
         definitionFunctionₘ
```

Examples of use:

```
f x y = (a+1) * (a+2)
      where a = (x + y) / 2

f x y = g (a+1) (a+2)
      where
        a = (x + y) / 2
        g x y = x * y

f x y = g (a+1) (a+2)
      where
        a = (x + y) / 2 ; g x y = x * y
```

let The expression `let` is also used for local definitions. The general syntax is:

```
f x₁ x₂ ... xₙ =
     let  definitionFunction₁
          ...
          definitionFunctionₘ
     in  exp
```

Examples of use:

```
f1 = let a = 3 + 2
    in a * a * a

f2 = 4 * (let a = 9 in a + 1) + 2
```

Additionally, `let` expressions can be used in GHCi to define functions which can be used during the current session. For instance:

[4]The scope of these internal functions is the equation of the function where this function is defined.

```
Prelude> let x = (2+)
Prelude> let y = x 4 + 3
Prelude> y
9
```

5. Currying and partial application

There are two alternatives to define a function with two or more arguments in Haskell. Lets take a look at two definitions of a summing operation: `add` and `cAdd`:

```
add :: (Int, Int) -> Int
add (x,y) = x + y
```

```
cAdd :: Int -> Int -> Int
cAdd x y = x + y
```

There exists an isomorphism between the domains of these two versions of the function.[5] The second version of the function is curried and has some advantages w.r.t. the first version of the function from the programming point of view:

- the number of parenthesis are reduced since to call the function we can write "`cAdd x y`" instead of `add(x,y)`.
- it facilitates the partial application of a function, that consist in not providing all the arguments to a function.

Partial function application works as follows: given a function `f` with two input arguments of types `a` and `b`, respectively, and output of type `c`:

```
   f :: a -> b -> c
```

Now suppose we have an expression `expr` of type `a` (written `expr::a`), that can be seen as a constant function of type `a`. We can write the expression "`f exp`" which is of type

```
   f exp :: b -> c
```

Meaning that `f expr` is another function that takes one argument of type `b` and returns an output of type `c`.

Let us look at another example with arithmetic operators. Suppose we have the following operator for multiplying (note that, since the operator is defined between parenthesis, we can use it in infix notation):

```
(*) :: Int -> Int -> Int
```

[5]An isomorphism (or bijective homomorphism) is a relation between algebraic structures that preserve the structure. Evaluation done in any of the domains are equivalent.

Now, it is possible to use it to define the following arithmetic functions:

```
squarepow :: Int -> Int
squarepow x = x * x

doubleHO :: (Int -> Int) -> Int -> Int
doubleHO f x = f (f x)

fourthpow :: Int -> Int
fourthpow = doubleHO squarepow
```

The function `doubleHO` is a higher order function because one of its arguments which is another function (the parameter `f`). It is important to remark that the function `fourthpow` expects an argument of type `Int` although its definition does not explicitly include a formal parameter (like for example the `x` in `fourthpow x`). The reason is that `doubleHO` is a higher order function that, after receiving `squarepow` as a first argument, leaves us with a function of type `Int -> Int`.

The operator " `->` " is right associated, meaning that `a -> b -> c` is equivalent to `a -> (b -> c)` and differs from `(a -> b) -> c`.

The function application operator is left associated, meaning that `f a b` is equivalent to `(f a) b` and differs from `f (a b)`.