Haskell Tutorial: Functions

May 27, 2020

[1]:
```
:opt no-lint
```

1 Functions

We have already seen how we can define simple functions. To define a function, we usually define the type signature (although this is optional) and afterwards we define the function itself.

Lets define a function to calculate the euklidian distance between two points $(x1, y1)$ and $(x2, y2)$. We represent the points naturally as tuples and we define a helper function to calculate the square of a number.

[2]:
```
square :: Double -> Double
square x = x * x

distance :: (Double, Double) -> (Double, Double) -> Double
distance (x1, y1) (x2, y2) = sqrt (square (x2 - x1) + square (y2 - y1))

distance (0,0) (4,3)
```

```
5.0
```

If we do not want the helper function to be used outside the definition of our `distance` function, we can define the helper function in a `where` clause.

[3]:
```
distance :: (Double, Double) -> (Double, Double) -> Double
distance (x1, y1) (x2, y2) = sqrt (square (x2 - x1) + square (y2 - y1))
       where
           square :: Double -> Double
           square x = x * x

distance (0,0) (4,3)
```

```
5.0
```

1.1 Pattern Matching

While defining functions, it is often necessary to distinguish difference cases. There are several ways to achieve this in Haskell. A simple but also very powerful mechanism for this is pattern matching: We match the arguments passed to a function against pattern and we apply the first matching function definition. Note that the order of the pattern matters (since we apply the definition of the first matching pattern) and that the set of pattern should catch all possibilities.

Lets define a function `vowel`, which returns `True` if an English language character is a vowel and `False` otherwise. Since the set of vowels is small, we can write a function definition for each vowel and one for the case where the character is not a vowel.

[4]:
```
vowel :: Char -> Bool
vowel 'a' = True
vowel 'e' = True
vowel 'i' = True
vowel 'o' = True
vowel 'u' = True
vowel _ = False

filter vowel "Hello World"
filter (not . vowel) "Hello World"
```

```
"eoo"
```

```
"Hll Wrld"
```

In the above example, we match the character passed to the function `vowel` against constants. The last clause uses the underscore `_`, a pattern used in situations where one wants to match anything but not use the matched value. The `vowel` function can be used as a filter function to extract all vowels of a string (which is a list of characters). By composing the `vowel` function with the `not` function, we can also easily extract all non-vowels of a string. We will come back to function composition later.

Here is another pattern matching example demonstrating how patterns can be used to match lists. Lets assume we want to define a function `elems` to obtain a string that says "no elements" if a list is empty, "one element" if a list has exactly one element, and "multiple elements" if the list has more than one element. We can achieve this using pattern matching by first testing the special cases and having the catch all case last.

[5]:
```
elems :: [a] -> [Char]
elems []     = "no elements"
elems (_:[]) = "one element"
elems _      = "multiple elements"

"The list has " ++ elems []
"The list has " ++ elems [[]]
"The list has " ++ elems [[],[]]
```

```
"The list has no elements"


"The list has one element"


"The list has multiple elements"
```

The first pattern tests whether the list is empty. The second pattern tests whether the list contains a head element followed by an empty list. The third patter matches all arguments and it serves as a catch-all pattern. Note that this definition works with infinite lists. An implementation that counts the list elements would be inferior.

Pattern matching can also be used to extract specific elements of a tuple. Lets define fst', snd', and trd' for 3-tuples. The following definitions are polymorphic, they work regardless which data types are present in a 3-tuple.

[6]:
```haskell
fst' :: (a, b, c) -> a
fst' (x, _, _) = x

snd' :: (a, b, c) -> b
snd' (_, y, _) = y

trd' :: (a, b, c) -> c
trd' (_, _, z) = z

fst' ("lost", 42, "number")
trd' (True, 1, "yes")
```

```
"lost"


"yes"
```

1.2 Recursion

Compared to imperative programming languages such as C, C++, or Java, a pure functional language like Haskell does not have constructs to program loops. Instead, we use *recursion* to implement loops. Recursion essentially means that the definition of a function may refer to itself. The idea is that a given problem is transferred into a slightly simpler problem until the problem is so simple that it can be directly solved. Hence, a recursive function definition typically consists of a simple case where the result can be directly provided plus one or more complex cases that can be transformed into simpler cases. This essentially means that we need a mechanism to distinguish the simple case from the more complex one.

Number sequences are classic examples for recursive functions and the probably most widely used recursively defined number sequence in computer science textbooks is the fibonacci series.

Fibonacci numbers are defined as follows (for $n \geq 0$):

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

[7]:
```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

map fib [0..10]
```

```
[0,1,1,2,3,5,8,13,21,34,55]
```

Recursion is particularly interesting for inductively defined data structures such as lists. A function to sum up the numbers in a list of numbers could be written as follows. (You will later see that this can be done even simpler.) The basic idea is to think of this function as defined as follows:

$$sum'(x[0..n]) = \begin{cases} 0 & \text{empty list} \\ x[0] + sum'(x[1..n]) & \text{otherwise} \end{cases}$$

Note how pattern matching is used to separate the head from the tail of the list in the general case. This is a very common pattern when functions process lists.

[8]:
```
sum' :: Num a => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs

sum' [1..10]
```

```
55
```

The standard `map` function applies a function to all elements of a list. Lets define our own `map'` function which does the same. The simple case is an empty list where we return an empty list. The general case is a list where we apply the function to the first element and construct a new list out of the new value returned by the function and the application of the function to the tail of the list.

[9]:
```
map' :: (a -> b) -> [a] -> [b]
map' f [] = []
map' f (x:xs) = (f x) : map' f xs

map' fib []
map' fib [0..10]
```

```
[]
```

```
[0,1,1,2,3,5,8,13,21,34,55]
```

1.3 Guards

While pattern are already a convenient mechanism, they have their limitations. If you want to test whether a more general condition is true or false, you can use guards. Guards are essentially expressions returning a boolean value that are evaluated in order of their appearance. The first expression evaluating to `True` determines the function definition that is used. We can rewrite the function producing fibonacci numbers using guards as follows:

[10]:
```
fib' :: Integer -> Integer
fib' n | n == 0    = 0
fib' n | n == 1    = 1
fib' n | n > 1     = fib' (n-1) + fib' (n-2)
fib' n | otherwise = error "fib' not defined for negative numbers"

map fib' [0..10]
fib' (-1)
```

```
[0,1,1,2,3,5,8,13,21,34,55]
```

```
fib' not defined for negative numbers
```

We can now handle the situation where the argument is negative. Instead of going into an infinite recursion, we call the `error` function. Note that `error` should only be used in situations where an error indicates a programming error. The `error` function should not be used to signal runtime errors. In fact, a good Haskell programmer prefers pure functions that have no side effects. Calling `error` is a pretty strong side effect since the call will abort the program execution. We will later learn about better mechanisms to handle runtime errors.

Since pattern are often more concise and thus more readable than guards, it is sometimes a good idea to mix guards and pattern. Here is another attempt to define a function producing fibonacci numbers:

[11]:
```
fib'' :: Integer -> Integer
fib'' n | n < 0 = error "fib'' not defined for negative numbers"
fib'' 0 = 0
fib'' 1 = 1
fib'' n = fib'' (n-1) + fib'' (n-2)

map fib'' [0..10]
fib'' (-1)
```

```
[0,1,1,2,3,5,8,13,21,34,55]
```

```
fib'' not defined for negative numbers
```

1.4 Where Bindings

Where bindings can be used in function definitions to bind names to constants, to define local helper functions, or to perform pattern matching. Lets again look at calculating the distance between two points. This time, we use where bindings to introduce constants that are used in the function definition.

[12]:
```
distance :: (Double, Double) -> (Double, Double) -> Double
distance (x1, y1) (x2, y2) = sqrt (dx^2 + dy^2)
    where dx = x2 - x1
          dy = y2 - y1

distance (0,0) (4,3)
```

```
5.0
```

It is also possible to do pattern matching in where bindings.

[13]:
```
distance :: (Double, Double) -> (Double, Double) -> Double
distance x y = sqrt ((x2 - x1)^2 + (y2 - y1)^2)
    where (x1, y1) = x
          (x2, y2) = y

distance (0,0) (4,3)
```

```
5.0
```

Finally, we can define helper functions in where bindings that are not accessible outside the function definition. While where bindings may improve readability and may help to reduce duplication of code, they may also reduce readability if they are used too much.

[14]:
```
distance :: (Double, Double) -> (Double, Double) -> Double
distance (x1, y1) (x2, y2) = sqrt (sq dx + sq dy)
    where sq :: Double -> Double
          sq x = x * x
          dx = x2 - x1
          dy = y2 - y1

distance (0,0) (4,3)
```

```
5.0
```

1.5 Let Bindings

Let bindings are similar to where bindings. Since let bindings are expressions, they can appear anywhere where expressions can appear. However, different to where bindings, let bindings cannot be referenced in guards.

[15]:
```
distance :: (Double, Double) -> (Double, Double) -> Double
distance (x1, y1) (x2, y2) =
    let dx = x2 - x1
        dy = y2 - y1
    in sqrt (dx^2 + dy^2)

distance (0,0) (4,3)
```

```
5.0
```

Let bindings can also do pattern matching.

[16]:
```
distance :: (Double, Double) -> (Double, Double) -> Double
distance x y =
    let (x1, y1) = x
        (x2, y2) = y
    in sqrt ((x2 - x1)^2 + (y2 - y1)^2)

distance (0,0) (4,3)
```

```
5.0
```

1.6 Case Expressions

Case expressions are expressions where the expression evaluated is selected by a value (yielded by another expression) matched against a sequence of pattern. Lets look at an example. The value returned by the expression `n` is matched against the pattern `0`, `1`, and `otherwise` and depending on the match, either the expression `0`, `1`, or `fib (n-1) + fib (n-2)` is evaluated.

[17]:
```
fib n = case n of 0 -> 0
                  1 -> 1
                  otherwise -> fib (n-1) + fib (n-2)

map fib [0..10]
```

```
[0,1,1,2,3,5,8,13,21,34,55]
```

We can also rewrite the `sum'` function from above using a case expression.

[18]:
```
sum' :: Num a => [a] -> a
sum' xs = case xs of [] -> 0
```

```
                    (x:xs) -> x + sum' xs

sum' [1..10]
```

```
55
```

Case expressions tend to be relatively verbose and hence most Haskell programmers try avoid them if other syntactic constructs are sufficient and lead to shorter and more elegant and readable code. However, there are situations where resorting to case expressions is a benefit.

1.7 Function Composition

It is possible to create new functions by combining existing functions. Function composition is denoted using the `.` operator. (We have already used function composition once at the very beginning of this notebook.) In general, function composition means that the function on the right hand side of the `.` operator is applied first and then the function on the left hand side of the `.` operator. In other words, `f . g` results in the application of `g` followed by the application of `f`. Obviously, the types of the functions need to match. The type of the `.` operator is `(.) :: (b -> c) -> (a -> b) -> a -> c`.

[19]:
```
filter (not . vowel) "Hello World"
```

```
"Hll Wrld"
```

1.8 Lambda Functions

Lambda functions, also called anonymous functions, are helper functions without a name. Haskell's foundation is a universal model of computation called Lambda Calculus. A backslash character is commonly used to define lambda functions because of its visual resemblance with the Greek letter lambda (λ).

[20]:
```
(\x -> x*x) 2
```

```
4
```

Lambda functions behave like regular functions with names. However, a lambda function can only have a single clause in its definition. Hence, we must be sure that our pattern covers all cases, otherwise runtime errors will occur. For example, the following definition of `tail'` will be unsafe for an empty list.

[21]:
```
tail' :: [t] -> [t]
tail' = \(_:xs) -> xs

tail' [1..10]
```

```
[2,3,4,5,6,7,8,9,10]
```

Lambda functions are useful in situation where a function is needed that can be easily defined inline. For example, lets multiply all integers or a list by two and add one.

[22]:
```
map (\x -> x * 2 + 1) [1..10]
```

```
[3,5,7,9,11,13,15,17,19,21]
```

A lambda function gives us a very concise and readable solution without having to define a helper function and finding a good name for it. Of course, we could solve the same task using other means, for example, using function composition.

[23]:
```
map ((+ 1) . (* 2)) [1..10]
```

```
[3,5,7,9,11,13,15,17,19,21]
```

1.9 Partial Functions and Currying

In Haskell, all functions take only a single argument. This may sound like a contradiction since we are meanwhile used to functions will multiple arguments. To resolve what may appear as a contradiction, we need to look at type signatures again.

[24]:
```
take' :: Int -> [a] -> [a]
take' 0 xs     = []
take' n (x:xs) = x : take (n-1) xs
```

The function `take'` has in its signature only one parameter (i.e., an `Int`). After the `Int` argument has been applied, a function is returned that takes a list of some type `a` as the argument and which returns a list of the same type `a`. We can make use of this when we need a function `take3` that takes the first three elements of a list. We define `take3` to be the `take` function with the first argument fixed to `3`.

[25]:
```
take3 :: [a] -> [a]
take3 = take' 3

take3 [1..10]
```

```
[1,2,3]
```

We have already used currying when we created a `+1` function or a `*2` function.

[26]:
```
map ((+ 1) . (* 2)) [1..10]
```

```
[3,5,7,9,11,13,15,17,19,21]
```

From a mathematical point of view, currying a function expression means successively splitting away arguments from the right to the left. In the general case of currying an m-ary function expression: $f(x_1, x_2, \ldots, x_m) = f^{m-1}(x_1)(x_2)\ldots(x_m)$.