Curried functions and other topics

In this section we discuss some more advanced topics related to Haskell's typing model.

We start with as discussion of the process of currying function parameters, which is possible because of the dominant nature of function values in functional programming. We then move on to discuss type variables which enable *generic programming* techniques. We end with a discussion of type classes and their significance.

Curried functions

Looking at the example of the range function above:

```
range :: (Int, Int) -> [Int]
range (a, b) = [a..b]
```

You may be tempted to think of this function as having as input *two parameters*, the a and the b. In reality it has only *one parameter*, namely the *tuple* (a, b). This is why the type for the function has one thing on the left side of the arrow, namely the compound type (Int, Int).

This is an important step: Compound types allow us the illusion of multiple parameters when in reality there is only one parameter.

There is however one other way of allowing multiple parameters, which is called *currying* in honor of Haskell Brooks Curry once again. The main idea is that functions can be specified to take *multiple parameters one at a time.* An example is in order, using the function take we saw earlier. A typical call to take would look like this:

```
take 3 [1..10]
```

So we are calling take, providing it with two parameters, and get back the result list.

However, the "curried" nature of the function lies in the fact that we could provide only the first argument, and thus create a new function that simply expects a list as input:

```
prefix = take 3           -- prefix is now a function
prefix [1..10]            -- This is the same as `take 3 [1..10]`
```

Providing only partial arguments to a curried function, and thus effectively creating a new function, is an extremely common practice, and the system is built so that this process is very efficient.

Let us look at another example:

```
f x y = x + y                 -- function of two variables
add3 = f 3                    -- new function
add3 10                       -- same as f 3 10
```

Types for carried functions

A curried function is basically *a function whose return value is again a function.* When we write f x y = x + y what Haskell reads is:

> f is a function of one argument x, whose result is a new function of one argument y, whose result is adding the x to the y.

So Haskell reads f x y as:

```
(f x) y
```

In other words, f is applied to x and returns a function. That function is then applied to y to get us the result.

This helps us understand the type of such a function:

```
f :: Int -> (Int -> Int)
```

Since these functions are so common, it is customary to omit the parentheses: *Arrow types are right-associative.*

Practice. Determine the types for the following functions. Do not worry about implementing the functions, you just need to determine their type.

1. take from the standard functions. Assume the elements in the list are integers.
2. drop from the standard functions. Assume the elements in the list are integers.
3. hasEnough from the previous notes. Assume the elements in the list are integers.
4. isSubstring: Given a string and another string, it returns whether the first string is contained somewhere within the second string.
5. max3: Given three numbers, returns the maximum of the three.
6. evaluate: This function is called with two (curried) arguments. The first argument is a function f that takes as input an integer, and returns as output an integer. The second argument is an integer. The result is what happens when we apply f to that second argument.

More examples of curried functions

We will discuss in this section some more examples of curried functions. We will study these functions and more later.

zip zip is a function that takes two lists and groups them pairwise:

```
zip [1,2,3] ['a', 'b', 'c'] = [(1, 'a'), (2, 'b'), (3, 'c')]
```

We can provide zip with only its first argument:

```
enumerate = zip [1..]
-- Calling enumerate numbers the elements of the list we give it:
enumerate "hey now!"
```

Let's construct the type of the function zip. We start with its first argument: It expects a list as its first argument:

```
zip :: [t] -> ....
```

What is returned if we provide just the first argument is now a function that expects the second argument, which is another list:

```
zip :: [t] -> ([s] -> ...)
```

Finally, the function returns tuples formed out of elements of the first list and the second list, so those tuples have type (t, s). Therefore we end up with the following type for zip:

```
zip :: [t] -> ([s] -> [(t, s)])
-- usually written as:
zip :: [t] -> [s] -> [(t, s)]
```

map map is a function that takes as arguments a function and a list, and it applies the function to each element of the list and creates a new list in the process:

```
times2 x = x * x
map times2 [2, 3, 4] -- results in [4, 9, 16]
```

We can create a new function by providing just the function part to the map function:

```
square = map times2
square [1, 2, 3, 4, 5] --  results in [1, 4, 9, 16, 25]
cube = map (\x -> x * x * x)
-- toUpper is a function Char -> Char
import Data.Char (toUpper)
-- This makes stringToUpper a function String -> String
stringToUpper = map toUpper
stringToUpper "hello there!"  -- result is "HELLO THERE!"
```

Let us now work out the type of map. It is a function that takes as input a function:

```
map :: (... -> ...) -> (...)
```

That first argument function must have some input and output types:

```
map :: (a -> b) -> (...)
```

Now map takes a second argument, which is in fact a list to whose elements we can apply the first argument function:

```
map :: (a -> b) -> ([a] -> ...)
```

And finally it returns a list made out of the results of applying our function:

```
map :: (a -> b) -> ([a] -> [b])
-- usually written as:
map :: (a -> b) -> [a] -> [b]
```