Advanced Haskell Typing concepts

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.

Reading

- Sections 3.7-3.9
- Practice exercises (3.11): 1, 2, 3, 4, 5

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]
```

Polymorphism

Polymorphism is a general term describing how the same piece of code might behave differently depending on the arguments provided. The term usually refers to a function or operator call. There are fundamentally two different kinds of polymorphism:

Parametric Polymorphism refers to the situation where the same function code may act on values of different types but without changing the code. In that case the type of the corresponding value is a "parameter". A good example of this is functions operating on lists: The function `head` does not particularly care what type of values your list contains, only that it contains a list. So it can operate on *any list type*, and the content type of the list type is in effect a parameter.

In C++ parametric polymorphism is achieved via *templates*, and in Java via *generics* (`<T>`).

Ad-hoc Polymorphism refers to the situation where the same function symbol refers to multiple code chunks, and the decision on which code chunk to execute depends on the types of the arguments. A good example of this is the addition operation `+` or the equality operator `==`. Testing if two integers are equal requires a different code than testing if two strings are equal, yet they are both written the same way.

In C++ and Java you have probably encountered this as function/operator overloading, where the meaning of an expression like `a.add(b)` depends on the types of `a` and `b`, and different functions will be executed depending on those types. One can further say that object-oriented programming requires this kind of polymorphism in an essential way, to direct method calls to the correct place.

Almost every functional programming language implement parametric polymorphism in a similar way, via *type variables*. Support for ad-hoc polymorphism varies. Some languages don't have it at all, and those that do implement it in different ways. Haskell uses a clever concept called *type classes* that we will discuss in a moment.

Parametric Polymorphism: Type Variables

Some functions can do their job perfectly fine without needing to know precisely the type of value that they act on. A good example of that is the `tail` function for lists. A possible implementation for it could be simply:

```
drop (x:xs) = xs
```

If Haskell tries to determine the type of this function, it will run into some trouble. It can tell that the input must be a list, and that the output must be a list *of the same type*. But it has no way of knowing what type of elements the list contains, nor does it care; it can do its job regardless.

In order to assign a proper type to this function, we must expand our type system. We don't want a drop function that only works on say [Int] values, nor one that only works on [Char] values. We need a function that only works on any [...] values.

This is where *type variables* come in. A **type variable** is essentially a variable used to represent a type in a function signature. These are lowercase and typically consist of only one letter, like a, b, t. When a function is being called, these types are **instantiated** for a particular type.

As an example, the type of the drop function would therefore be:

```
drop :: Int -> [t] -> [t]
```

This says that drop accepts as input a value of any list type, and returns a value of *the same list type*. When the function is actually used, like in

```
drop 1 [1, 2, 3]    -- t=Int. Used as drop :: [Int] -> [Int]
drop 1 "abc"        -- t=Char. Used as drop :: [Char] -> [Char]
```

a specific type is chosen in place of the variable, for each use of the function. But the body of the function that is executed does not change.

If the body of the function does not provide any constraints on the types of some of the parameters, the function typically ends up with a parametric type (unless the programmer specified a more stringent type).

We can also have *multiple type variables*, if there are elements in the function that have arbitrary but different types. An example is the zip function. It takes two lists of elements and returns a new list by forming pairs from elements one from each list. It looks roughly like this:

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

In this case we don't care what the types of the two lists are, and they can be different from each other. But the type of the result list depends on them. We could write the type for zip thus:

```
zip :: [a] -> [b] -> [(a, b)]
```

Practice: Determine the types of the following functions:

1. head, tail.
2. take.
3. length.
4. fst. This function takes as input a tuple of two elements and returns the first element.

Ad-hoc Polymorphism: Overloaded Types and Type Classes

Ad-hoc polymorphism is a bit trickier, especially in a language that performs type inference, as the system must be able to see an expression like `x+y` and infer some type information regarding `x` and `y`. This is accomplished by a couple of related ideas, namely *overloaded types* (often refered to as *bounded polymorphism*) and *type classes*.

A **overloaded type** is a type that comes with a certain constraint. For instance the type of an `add` function may look like this:

```
add :: Num t => t -> t -> t
add x y = x + y
```

What this tells us that is that the function add takes two arguments of a certain type and returns a value of that same type, but it can't just be any type. It has the constraint `Num t`, which says that it must be a "number type".

Even the type of a single number by itself has a similar constraint, because that number can be thought of as one of the many number types:

```
3 :: Num t => t
```

These constraints come from the so-called type-classes: A **type class** is a list of specifications for operations on a type. An **instance** of a type class is a specific type along with definitions for these operations.

A good example of a type-class is the `Num` type class for numbers. Any instance of this class must provide implementations for the following functions:

```
-- The Num class. An instance Num a must implement:
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a          -- sign
```

If we wanted to, we could for instance make the `Char` type an instance of the Num class by specifying how each of these operations would work. From that point on we could be writing `'a' + 'b'` and the system won't complain.

Standard Type Classes Implementing your own type class is a more advanced feature. But there are many standard type classes that are in constant use, and we will see more as we move on. Here are some of the standard ones:

Num We already encountered this earlier. It contains the following functions:

```
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a
```

Eq The "equality" type class. Values of types that implement Eq can be compared to each other. This contains the following functions:

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

You can see a "type error" if you try to compare two functions, as function types are not instances of the Eq class:

```
(+) == (-)     -- Look at the error
```

Ord This represents ordered types. These are an extension of Eq, and in addition to those functions must also implement these:

```
(<) :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>) :: a -> a -> Bool
(>=) :: a -> a -> Bool
min :: a -> a -> Bool
max :: a -> a -> Bool
```

Show This represents types whose value have a string representation. These are the only values that Haskell will print out for you without complaining. They need to implement a single function:

```
show :: a -> String
```

Read This represents types that know how to turn a string into a value. They need to implement a single method:

```
read :: String -> a
```

Here's an example use of this, to read in a tuple from a string representation:

```
read "(True, 5)" :: (Bool, Int)    -- We must specify the return type.
```

Integral

This is an extension of the Num class. It further requires the implementation of integer division operations:

```
div :: a -> a -> a
mod :: a -> a -> a
```

Fractional This is an extension of the Num class that supports fractional division and reciprocation:

```
(/) :: a -> a -> a
recip :: a -> a
```

Many of these type classes extend to compound types if there is a specification on how to do so. For example tuples are instances of the class Ord as long as their components are, and the same for lists:

```
(3, 4) > (2, 5)
[3, 4, 5] > [2, 5, 6, 7]
```

Practice: Figure out the types of the following functions, including type class specifications:

1. posDiff defined by posDiff x y = if x > y then x − y else y − x.

2. maxList defined on lists by:

```
maxList (x:[]) = x
maxList (x:xs) = if x > restMax then x else restMax
            where restMax = maxList xs
```

3. has that checks for the existence of an element in a list, and is defined by:

```
has elem []       = False
has elem (x:rest) = elem == x || has elem rest
```