Lecture 7

Data types

We have already seen an example of a compound data type namely list. Recall that, a list is either an empty list or a list with a head element and rest of the list. We begin by defining a list data type. Haskell already provides a list data type so we do not need to define a user defined data type. However, we do this for illustration

```
> import Prelude hiding (sum) -- hide the standard sum function
> data List a = EmptyList
>              | Cons  a (List a)
```

One reads this as follows "List of `a` is either `EmptyList` or a `Cons` of `a` and `List` of `a`". Here the variable `a` is a type variable. The result of this is that `List` is now a polymorphic data type. We can instatiate this with any other Haskell data types. A list of integers is then `List Integer`.

The identifiers `EmptyList` and `Cons` are the two *constructors* of our new data type `List`. The constructors can now be used in Haskell expressions. For example `EmptyList` is a valid Haskell expression. So is `Cons 2 EmptyList` and `Cons 1 (Cons 2 EmptyList)`. The standard list actually has two constructors, namely `[]` and `(:)`.

7.1 Pattern Matching

We can now define functions on our new data type `List` using pattern matching in the most obvious way. Here is a version of `sum` that works with `List Int` instead of `[Int]`

```
> sum :: List Int -> Int
> sum EmptyList   = 0
> sum (Cons x xs) = x + sum xs
```

As the next example, we two functions to convert from our list type to the standard list type.

```
> toStdList   :: List a -> [a]
> fromStdList :: [a] -> List a

> toStdList EmptyList   = []
> toStdList (Cons x xs) = x : toStdList xs

> fromStdList []     = EmptyList
> fromStdList (x:xs) = Cons x (fromStdList xs)
```

1. **Exercise**: Define the functions `map` `foldr` and `foldl` for our new list type.

7.2 Syntax of a data type

We now give the general syntax for defining data types.

```
data Typename tv_1 tv_2 tv_n = C1 te_11 te_12 ... te_1r1
                             | C2 te_21 te_22 ... te_2r2
                             |      . . .
                             | Cm te_m1 te_m2 ... te_mrm
```

Here data is a key word that tells the complier that the next equation is a data type definition. This gives a polymorphic data type with *n type arguments* `tv_1,...,tv_n`. The `te_ij`'s are arbitrary type expressions and the identifiers `C1` to `Cm` are the constructors of the type. Recall that in Haskell there is a constraint that each variable, or for that matter type variable, *should* be an identifer which starts with a lower case alphabet. In the case of type names and constructors, they *should* start with upper case alphabet.

7.3 Constructors

Constructors of a data type play a dual role. In expressions they behave like functions. For example in the `List` data type that we defined the `EmptyList` constructor is a constant List (which is the same as 0-argument function) and `Cons` has type `a -> List a -> List a`. On the other hand constructors can be used in pattern matching when defining functions.

7.4 The Binary tree

We now look at another example the binary tree. Recall that a binary tree is either an empty tree or has root and two children. In haskell this can be captured as follows

```
>
> data Tree a = EmptyTree
>             | Node (Tree a) a (Tree a)
>
```

To illustrate function on tree let us define the `depth` function

```
> depth :: Tree a -> Int
> depth EmptyTree           = 0
> depth (Node left _ right) | dLeft <= dRight = dRight + 1
>                           | otherwise       = dLeft  + 1
>       where dLeft  = depth left
>             dRight = depth right
```