Functional Programming

Paul Mellies

Lesson 18 : Parameterized types
Parameterized types which take types as arguments

The most basic parameterized type one can make is the type `Box` defined here:

```haskell
data Box a = Box a deriving Show
```

The `Box` type is an abstract container which can hold any type.
Parameterized types which take types as arguments

The parameterized type `Box` behaves in the following way:

```ghci
ghci> let n=6 :: Int
ghci> :type Box n
Box n :: Box Int
ghci> let word = "hello"
ghci> :type Box word
Box word :: Box [Char]
ghci> let f x = x
ghci> :type Box f
Box f :: Box (t -> t)
ghci> let otherBox = Box n
ghci> :type Box otherBox
Box otherBox :: Box (Box Int)
```

If we think of types as logical propositions, the parameterized type `Box` turns types into types and thus plays the same role as the modality □ of a modal logic like S4.
Parameterized types which take types as arguments

We can also define simple functions such as the \texttt{wrap} function

\begin{verbatim}
wrap :: a -> Box a
wrap x = Box x
\end{verbatim}

and the \texttt{unwrap} function, defined by pattern matching:

\begin{verbatim}
unwrap :: Box a -> a
unwrap Box x = x
\end{verbatim}

to put items inside a box, or take them outside of a box.
The parameterized type \texttt{List} of lists

The most common parameterized type is the \texttt{List} type.

The following definition of the built-in list \texttt{[]} appears when one writes \texttt{:info []} in the interactive mode of Haskell:

\begin{center}
\textbf{data \texttt{[}a\texttt{]} = \texttt{[]} \mid a:\texttt{[}a\texttt{]}}
\end{center}

Note that the notation \texttt{[a]} is syntactic sugar in Haskell for the type \texttt{[]}\texttt{a}. 
**Type classes and polymorphism**

It is also possible to create the parameterized `List` type in the following way:

```haskell
data List a = Empty | Cons a (List a) deriving Show
```

We can define a list of integers in the built-in list type or in the `List` type:

- **Built-in list example:**
  ```haskell
  builtinEx1 :: [Int]
  builtinEx1 = 1:2:3:[]
  
  ourListEx1 :: List Int
  ourListEx1 = Cons 1 (Cons 2 (Cons 3 Empty))
  ```

- **Our list example:**
  ```haskell
  builtinEx2 :: [Char]
  builtinEx2 = 'c':'a':'t':[]
  
  ourListEx2 :: List Char
  ourListEx2 = Cons 'c' (Cons 'a' (Cons 't' Empty))
  ```

Similarly with a list of characters:
The map function

We have seen in Lesson 9 that there is a map function in the Prelude library

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

The map function enables one to apply a function of type

\[
\text{function} :: a \rightarrow b
\]

to every element of a list of type \([a]\).
The map function

Typically, the map function applied to the function

```haskell
square :: Int -> Int
square x = x * x
```

and to a list of integers produces a list of integers:

```haskell
ghci> map square [1..10]
[1,4,9,16,25,36,49,64,81,100]
```

Similarly, the map function applied to the function

```haskell
show :: Int -> String
```

and to a list of integers produces a list of strings:

```haskell
ghci> map show [1..10]
["1","2","3","4","5","6","7","8","9","10"]
```
The map function revisited

The map function may be adapted to the type List by defining the function

\[
\text{ourMap} :: (a \rightarrow b) \rightarrow \text{List} \ a \rightarrow \text{List} \ b
\]

This is done by pattern matching, in the following way:

\[
\text{ourMap} \ _ \ \text{Empty} = \text{Empty} \\
\text{ourMap} \ \text{func} \ (\text{Cons} \ x \ \text{xs}) = \text{Cons} \ (\text{func} \ x) \ (\text{mapList} \ \text{func} \ \text{xs})
\]

The ourMap function typically behaves in the following way:

\[
\text{ghci}\text{> ourMap ourListEx1} \\
\text{Cons} \ 1 \ (\text{Cons} \ 4 \ (\text{Cons} \ 9 \ \text{Empty}))
\]
Parameterized types with parameter in a type class

In many situations, we want to use a parameterized type where the parameter \( t \) is not any type but a type \( t \) of a specific type class.

This is nicely illustrated by the \texttt{quicksort} function on lists, defined in this way:


def quicksort [] = []
def quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
    biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted

Note that we use list comprehension

\[
[a \mid a <- xs, a <= x] \quad \quad [a \mid a <- xs, a > x]
\]

to select the elements \texttt{smaller} and \texttt{larger} than the pivot \( x \) in the list \( xs \).
Parameterized types with parameter in a type class

When one asks for the type of the `quicksort` function, one obtains the answer:

```
ghci> :type quicksort
quicksort :: Ord t => [t] -> [t]
```

Here, the parameterized type `[t] -> [t]` indicates that the `quicksort` function is designed to transport lists of elements of type `t` into lists of elements of type `t`,

Moreover, the prefix `Ord t =>` indicates that the `quicksort` function can only be applied to a list whose elements have a type `t` in the class `Ord` of ordered types.

This makes sense: a list of elements can be sorted only when its elements are ordered!
Thank you for your attention!