Functional Programming

Paul Mellies

Lesson 11 : Type Basics
Type signature for a variable

Here is how one defines a variable \( x \) and assigns it the type `Int` in Haskell:

Note that all types in Haskell start with a capital letter in order to distinguish them from functions and variables which start with a lowercase letter.
The Int type of machine integers

Suppose the variable \( x \) of type \texttt{Int} is defined with the signature

\[
\begin{array}{l}
x :: \texttt{Int} \\
x = 2
\end{array}
\]

The type \texttt{Int} is the type of the \textbf{machine integers} of 32 or 64 bits. The computations below indicate the intrinsic limits of this type:

\[
\begin{array}{l}
\texttt{ghci}\> \texttt{x*2000} \\
4000 \\
\texttt{ghci}\> \texttt{x^2000} \\
0
\end{array}
\]

Haskell handles exceeding the bounds of \texttt{Int} by returning the number 0.

We will see in Lesson 13 that \texttt{Int} is in fact an \textbf{instance} of \textbf{bounded type} and is limited for that reason by a maximum and a minimum value.
The Integer type of integers

Suppose now that the variable \( y \) of type \textbf{Integer} is defined with the signature

\[
y :: \text{Integer} \\
y = 2
\]

One clearly sees the difference between the two integer types \textbf{Int} and \textbf{Integer} by repeating the same computations:

\begin{verbatim}
ghci> y*2000
4000
ghci> y^2000
114813069527425452423283320117768198402231770208869520047764273682576626139237031
385665948631650626991844596463898746277344711896086305533142593135616665318539129
98914531280000688779148240044871428926990063486244781615463646388363947317026040
466353970904996558162398808944629605623311649536164221970332681344168908984458505
602379484807914058900934776500429002716706625830522008132236281291761267883317206
598995396418127021779858404042159853183251540889433902091920554957783589672039160
081957216630582755380425583726015528348786419432054508915275783882625175435528800
822842770817965453762184851149029376
\end{verbatim}
The **Char**, **Double** and **Bool** types

Haskell supports all the basic types expected of a programming language:

- the type **Char** of characters, as witnessed in
  ```haskell
  letter :: Char
  letter = 'a'
  ```

- the type **Double** of floating point numbers, as witnessed in
  ```haskell
  interestRate :: Double
  interestRate = 0.375
  ```

- the type **Bool** of boolean values **True** and **False**, as witnessed in
  ```haskell
  isFun :: Bool
  isFun = True
  ```
The List types

Every type `AType` induces a type `[AType]` of lists of values of that type:

- the type `[Int]` of lists of integers, as witnessed in

  ```
  values :: [Int]
  values = [1,2,3]
  ```

- the type `[Double]` of lists of floating point numbers, as witnessed in

  ```
  testScores :: [Double]
  testScores = [0.99,0.7,0.8]
  ```

- the type `[Char]` of lists of characters, as witnessed in

  ```
  letters :: [Char]
  letters = ['a','b','c']
  ```
The **String** type

A **string** is the same thing as a **list of characters**.

**Illustration:** the variable `letters` defined as the list of characters `['a', 'b', 'c']`

```
letters :: [Char]
letters = ['a','b','c']
```

is in fact equal to the string "abc" as testified by the equality predicate:

```
ghci> letters == "abc"
True
```

It is often convenient to use the type **String** as a synonym for `[Char]`

```
aPet :: String
aPet = "dog"
```

is the same as

```
aPet :: [Char]
aPet = "dog"
```
The **Tuple** types

We have already encountered tuples in Lesson 4.

As we will see, **Tuple** types are very convenient for modeling simple data types.

**Illustration.** Three basic examples of type signatures based on **Tuple** types:

```
ageAndHeight :: (Int,Int)
ageAndHeight = (34,74)
```

```
firstLastMiddle :: (String,String,Char)
firstLastMiddle = ("Oscar","Grouch","D"")
```

```
streetAddress :: (Int,String)
streetAddress = (123,"Happy St.")
```
Function types

In Haskell, functions also have type signatures 😊

Typically, the type signature for the `double` function looks like this:

```
double :: Int -> Int
double n = n*2
```

The arrow \( \rightarrow \) separates the types of the arguments and of the return values.
In Haskell, functions also have type signatures 😊

Typically, the type signature for the `double` function looks like this:

```
double :: Int -> Int
double n = n*2
```

The arrow `->` separates the types of the arguments and of the return values.
Converting one type into another

Imagine that you wish to give the type signature

\[
\text{half :: Int} \to \text{Double}
\]

to the `half` function which divides an integer and returns a floating point number. In that case, the original definition

\[
\text{half} = \text{n}/2
\]

does not compile anymore, because division `/` expects two floating point numbers. For that reason, the original code has to be replaced by

\[
\text{half} = (\text{fromIntegral n})/2
\]

where the `fromIntegral` function converts the integer `n` into a double.

**Exercise.** Why can we keep the number 2 in the `fromIntegral` function as it is and not replace it by `fromIntegral 2`?
Converting values to strings

The `show` function converts values of `Int`, `Char` and `Double` types to strings.

The conversion works in a straightforward way:

```
ghci> show 6
"6"
ghci> show 'c'
"'c'"
ghci> show 6.0
"6.0"
```

We will see in Lesson 13 that the `show` function works for every value of a type which is an instance of the Show class. This is the case of `Int`, `Char` and `Double`. 
Converting strings to values

The `read` function works in the converse direction: it converts a string to a value. However, the conversion is not as simple as in the case of the `show` function. Indeed, in order to perform the operation

\[ z = \text{read } "6" \]

one needs to know the expected type `Int`, `Integer` or `Double` of the output:

\[
\begin{array}{l}
\text{ghci}> \text{read } "6" :: \text{Int} \\
6 \\
\text{ghci}> \text{read } "6" :: \text{Double} \\
6.0
\end{array}
\]

In many cases, this can be achieved by type inference: for instance, one can deduce from the expression `q=z/2` that the variable `z` should be treated as of `Double` type. In other cases, and more generally, it is a good idea to use type signatures.
Multi-argument functions

Suppose that you want to define a `makeAddress` function which takes

1. a house number
2. a street address
3. the name of a town

and then returns the address as a triple.

The type signature of the `makeAddress` function looks like this:

```
makeAddress :: Int -> String -> String -> (Int,String,String)
makeAddress number street town = (number,street,town)
```
Multi-argument functions

You can rewrite a multi-argument function as a sequence of nested lambda functions:

```plaintext
makeAddress number street town = (number, street, town)
makeAddressLambda = (number -> (street -> (town -> (number, street, town))))
```
Multi-argument functions

You could then call this function in that way:

```ghci
ghci> (((makeAddressLambda 123) "Happy St") "Haskell Town"
(123,"Happy St","Haskell Town")
```

In this format, each function returns a function waiting for the next!

Although this may look weird at first, if you think more about it:

```markdown
this is precisely how partial application works!!!
```

Note here the influence of the $\lambda$-calculus.
Multi-argument functions

As a matter of fact, the function

makeAddressLambda

is nothing but the desugared lambda version of the original function

makeAddress

In particular, the two functions are equivalent!

This explains why makeAddress can be called using partial applications:

ghci> (((makeAddress 123) "Happy St") "Haskell Town"
(123,"Happy St","Haskell Town")

and conversely why makeAddressLambda can be used as a ternary function:

ghci> makeAddressLambda 123 "Happy St" "Haskell Town"
(123,"Happy St","Haskell Town")
Type for first-class functions

Higher-order functions can take functions as arguments.

Typically, here follows the type signature for the `ifEven` function:

```haskell
ifEven :: (Int -> Int) -> Int -> Int
ifEven f n = if even n
    then f n
    else n
```

where `(Int -> Int)` in the type

```haskell
(Int -> Int) -> Int -> Int
```

indicates that the first argument expected by `ifEven` should be of type

`Int -> Int`
A case of parametric polymorphism

It is interesting to observe that the identity function

\[
\text{simple } x = x
\]

can be typed in different ways, for instance:

\[
\text{simpleInt} :: \text{Int} \rightarrow \text{Int} \\
\text{simpleInt } n = n
\]

for integers, or

\[
\text{simpleChar} :: \text{Char} \rightarrow \text{Char} \\
\text{simpleChar } c = c
\]

for characters. Can we find a better way to type the \text{simple} function?
**Type variables**

The idea is to use **type variables** indicated by lowercase letters such as \(a\), \(b\) or \(c\).

More remarkably, the identity function may be typed as:

\[
\begin{align*}
\text{simple} :: & \quad a \rightarrow a \\
\text{simple} & \quad x = x
\end{align*}
\]

Hence, when a Char is given as argument to the `simple` function

\[
\text{simple} \ 'h'
\]

the function behaves as though its type signature was

\[
\text{simple} :: \text{Char} \rightarrow \text{Char}
\]

and when a String is given as argument to the `simple` function

\[
\text{simple} \ "hello"
\]

the function behaves as though its type signature was

\[
\text{simple} :: \text{String} \rightarrow \text{String}
\]
Type variables

The type signature of a function may contain several type variables.

Consider for instance the `makeTriple` function

```
makeTriple :: a -> b -> c -> (a,b,c)
makeTriple x y z = (x,y,z)
```

which may be applied to a String, a Char and a String, in the following way:

```
nameTriple = makeTriple "Oscar" 'D' "Grouch"
```

In that case, the type signature which Haskell uses for `makeTriple` looks like:

```
makeTriple :: String -> Char -> String -> (String,Char,String)
```

All this is done internally and automatically in Haskell using **type inference**.
Thank you for your attention!