1 Types: Basics

We mentioned that Haskell has a static type system. The type of every expression is known at compile time, which leads to safer code. If you write a program where you try to divide a boolean type with some number, it won’t even compile. That’s good because it’s better to catch such errors at compile time. We can avoid such errors by defining the types of the variables we're working with:

```haskell
let x :: Int = 5
let y :: Int = 2
let z = x / y
```

Haskell will tell you that you can’t divide an integer by an integer, because it’s not defined for that type. That’s a compile-time error, which is far better than a runtime error like a division by zero.

We can also define functions that take types.

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

If you try to call `add` with an integer and a string, you’ll get a type error. That’s another compile-time error, which is better than a runtime error like a type mismatch.

```
add 5 "hello"
```

Haskell will tell you that you can’t add an integer and a string, because they’re not of the same type. That’s a compile-time error, which is far better than a runtime error like an incompatible operation.

The type system is based on types being either more specific or more general. For example, `Int` is a more specific type than `Number`, which is a more general type. The more specific types are more concrete and have less ambiguity. This means that Haskell can catch errors at compile time, which is far better than catching them at runtime. This leads to safer code.

```
let x :: Int = 5
let y :: Int = 2
let z = x / y
```

```
add 5 "hello"
```

```
add :: Int -> Int -> Int
add x y = x + y
```

```
import Data.Bits
f :: (Enum a) => a -> a
f x = x .&. 4
```

```
f 5
```

```
f False
```

```
f 0
```

```
import Data.Bits
f :: (Enum a) => a -> a
f x = x .&. 4
```

```
f 5
```

```
f False
```
time instead of having your program crash. Everything in Haskell has a type, so the compiler can reason quite a lot about your program before compiling it.

Unlike Java or Pascal, Haskell has type inference. If we write a number, we don’t have to tell Haskell it’s a number. It can infer that on its own, so we don’t have to explicitly write out the types of our functions and expressions to get things done. We covered some of the basics of Haskell with only a very superficial glance at types. However, understanding the type system is a very important part of learning Haskell.

A type is a kind of label that every expression has. It tells us in which category of things that expression fits. The expression True is a boolean, "hello" is a string, etc.

Now we’ll use ghci to examine the types of some expressions. We’ll do that by using the :t command: when followed by any valid expression, this command tells us its type.

```
ghci 1> :t 'a'
   'a' :: Char
```

```
ghci 2> :t True
   True :: Bool
```

```
ghci 3> :t "HELLO!"
   "HELLO!" :: [Char]
```

```
ghci 4> :t (True, 'a')
   (True, 'a') :: (Bool, Char)
```

```
ghci 5> :t 4 ≡ 5
   4 ≡ 5 :: Bool
```

We see that doing :t on an expression prints out the expression followed by :: and its type. :: is read as “if of type”. Note: explicit types are always denoted with the first letter in capital case.

- 'a' is of type Char
- True is of type Bool
- the type of "HELLO!" is [Char]; the square brackets denote a list, so we read that as being a list of characters
- unlike lists, each tuple length has its own type: so the expression (True, 'a') is of type (Bool, Char), whereas an expression such as ('a', 'b', 'c') is of type (Char, Char, Char)
- 4 ≡ 5 will always return False, so its type is Bool

Functions also have types. When writing our own functions, we can choose to give them an explicit type declaration. This is generally considered to be good practice except when writing very short functions.

From here on, we’ll give all the functions that we make type declarations. Remember the list comprehension we made previously that filters a string so that only caps remain? Here’s how it looks like with a type declaration.
removeNonUppercase has a type of `[Char] → [Char]`, meaning that it maps from a string to a string. That’s because it takes one string as a parameter and returns another as a result. The `[Char]` type is synonymous with `String`, so it’s clearer if we write `removeNonUppercase :: String → String`. We didn’t have to give this function a type declaration because the compiler can infer by itself that it’s a function from a string to a string (but we did anyway).

How do we write out the type of a function that takes several parameters? Here’s a simple function that takes three integers and adds them together:

```
ghci 9> let { addThree :: Int → Int → Int → Int;
    addThree x y z = x + y + z }
```

The parameters are separated with `→` and there’s no special distinction between the parameters and the return type. The return type is the last item in the declaration and the parameters are the first three.

Later on we’ll see why they’re all just separated with `→` instead of having some more explicit distinction between the return types and the parameters like `Int, Int, Int → Int` or something.

If you want to give your function a type declaration but are unsure as to what it should be, you can always just write the function without it and then check it with `:t`. Functions are expressions too, so `:t` works on them without a problem.

## 2 Common types

### 2.1 Int

`Int` stands for integer. It’s used for whole numbers. 7 can be an `Int`, but 7.2 cannot. `Int` is bounded, which means that it has a minimum and a maximum value; we’ll see these values on a 64-bit Linux machine very soon.

### 2.2 Integer

`Integer` also stands for integer. The main difference is that it’s not bounded so it can be used to represent really really big numbers. `Int`, however, is more efficient.
2.3  Float

Float is a real floating point with single precision.

2.4  Double

Double is a real floating point with double the precision.

2.5  Bool

Bool is a boolean type. It can have only two values: True and False.

2.6  Char

Char represents a character. It’s denoted by single quotes.

2.7  String

A list of characters [Char] is a String. It’s denoted by double quotes.
2.8 Tuples

Tuples are types but they are dependent on their length as well as the types of their components, so there is theoretically an infinite number of tuple types, which is too many to cover in this tutorial.

Note that the empty tuple () (called ‘unit’) is also a type, and it has only one value, symbolized the same way: ()

3 Type variables

What do you think is the type of the head function? head takes a list of any type and returns the first element, so what could it be? Let’s check:

ghci 17> :t head
head :: [a] → a

What is this a? It cannot be a type since types are always written with an initial capital letter. a is a type variable. That means that a can be of any type (this is much like generics in other languages).

Using type variables allows us to easily write very general functions – if the functions don’t use any specific behavior of the types in them.

Functions that have type variables are called polymorphic functions. The type declaration of head states that it takes a list of any type and returns one element of that type.

Although type variables can have names longer than one character, we usually give them names of a, b, c, d ...

Remember fst? It returns the first component of a pair. Let’s examine its type.

ghci 18> :t fst
fst :: (a, b) → a

We see that fst takes a tuple which contains two types and returns an element which is of the same type as the pair’s first component. That’s why we can use fst on a pair that contains any two types.

Note that just because a and b are different type variables, they don’t have to be different types, but they can. This is just as in classical first-order logic: two distinct variables don’t have to have different values relative to a variable assignment, but they can.

However, fst does require the type of the first component and the type of the return value to be the same.

4 Typeclasses: Basics

A typeclass is a sort of interface that defines some behavior. If a type is a part of a typeclass, that means that it supports and implements the behavior the typeclass describes.

For example, what is the type signature of the ≡ operator? The equality operator ≡ is a function; so are +, *, /, and pretty much all operators.

If a function is comprised only of special characters (like all the above operators), it’s considered an infix function by default. So if we want to examine its type, pass it to another function or call it as a prefix function, we have to put it in parentheses.

ghci 19> :t (≡)
(≡) :: Eq a ⇒ a → a → Bool
Everything before the ⇒ symbol is called a class constraint. We can read the previous type declaration like this: the equality function takes any two values that are of the same type and returns a \( \text{Bool} \). The type of those two values must be a member of the \( \text{Eq} \) class (this was the class constraint).

The \( \text{Eq} \) typeclass provides an interface for testing for equality. Any type where it makes sense to test for equality between two values of that type should be a member of the \( \text{Eq} \) class. All standard Haskell types except for \( \text{IO} \) (the type for dealing with input and output) and for functions are part of the \( \text{Eq} \) typeclass.

For example, the \( \text{elem} \) function has a type of \((\text{Eq } a) \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}\) because it uses \( \equiv \) over a list to check whether some value we’re looking for is in it.

```
ghci 20> :t elem
   elem :: Eq a ⇒ a → [a] → Bool
```

5 Common typeclasses

Figure 1: Haskell typeclasses (based on http://commons.wikimedia.org/wiki/File:Classes.svg).
5.1  Eq

Eq is used for types that support equality testing. The functions its members implement are \(\equiv\) and \(\not\equiv\). So if there’s an Eq class constraint for a type variable in a function, it uses \(\equiv\) or \(\not\equiv\) somewhere inside its definition.

All the types we mentioned previously except for functions are part of Eq, so they can be tested for equality.

```ghci
ghci 21> 5 \equiv 5
True
```

```ghci
ghci 22> 5 \not\equiv 5
False
```

```ghci
ghci 23> 'a' \equiv 'a'
True
```

```ghci
ghci 24> "Ho Ho" \equiv "Ho Ho"
True
```

```ghci
ghci 25> 3.432 \equiv 3.432
True
```

5.2  Ord

Ord is for types that have a natural ordering defined over them. All the types we covered so far except for functions are part of Ord. To be a member of Ord, a type must first have membership in the prestigious and exclusive Eq club – see Figure 1 above.

```ghci
ghci 26> :)hoogle -- info Ord
Prelude class Eq a \Rightarrow Ord a

The Ord class is used for totally ordered datatypes.
Instances of Ord can be derived for any user-defined datatype whose constituent types are in Ord. The declared order of the constructors in the data declaration determines the ordering in derived Ord instances. The Ordering datatype allows a single comparison to determine the precise ordering of two objects.

Minimal complete definition: either compare or <=. Using compare can be more efficient for complex types.
From package base class Eq a \Rightarrow Ord a

For example, Ord covers all the standard comparing functions such as >, <, \geq\ and \leq\.
5.2.1 The compare function and the Ordering type

The compare function takes two Ord members of the same type and returns an element of type Ordering.

Ordering is a type (not a typeclass) with only three values – GT, LT and EQ – meaning ‘greater than’, ‘less than’ and ‘equal’, respectively.

5.3 Show

Show says that its members can be presented as strings. All types covered so far except for functions are part of Show.

The most used function that deals with the Show typeclass is show: it takes a value whose type is a member of Show and presents it to us as a String.
5.4 Read

`Read` is sort of the opposite typeclass of `Show`: the associated function `read` takes a `String` and returns a type which is a member of `Read`.

```
ghci 36> show True
"True"
```

```
ghci 37> :t read
read :: Read a ⇒ String → a
```

```
ghci 38> read "True" ∨ False
True
```

```
ghci 39> read "8.2" + 3.8
12.0
```

```
ghci 40> read "5" − 2
3
```

```
ghci 41> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

All types covered so far are in this typeclass.
But what happens if we try to do just `read "4"`?

```
ghci 42> read "4"
```

`ghci` tells us that it doesn’t know what we want in return. Notice that in the previous uses of `read`, we did something with the result afterwards. That way, `ghci` could infer what kind of result we wanted out of our `read`. If we used it as a boolean, it knew it had to return a `Bool`.

But now, it only knows that we want some type that is part of the `Read` class, but it doesn’t know which one. Let’s take another look at the type signature of `read`.

```
ghci 43> :t read
read :: Read a ⇒ String → a
```

It returns a type that’s part of `Read` but if we don’t try to use it in some way later, it has no way of knowing which type. We can use explicit type annotations to help `ghci` figure it out.
5.4.1 Type annotations

Type annotations are a way of explicitly saying what the type of an expression should be. We do that by adding :: at the end of the expression and then specifying a type.

```
ghci 44> read "5" :: Int
5

ghci 45> read "5" :: Float
5.0

ghci 46> (read "5" :: Float) * 4
20.0

ghci 47> read "[1,2,3,4]" :: [Int]
[1,2,3,4]

ghci 48> read "[1,2,3,4]" :: [Double]
[1.0,2.0,3.0,4.0]

ghci 49> read "(3, 'a')" :: (Int, Char)
(3,'a')
```

In most cases, the interpreter / compiler can infer the type of an expression by itself. But sometimes it needs a little help from us, e.g., if it doesn’t know whether to return a value of type Int or Float for an expression like read "5".

5.5 Enum

Enum members are sequentially ordered types: these types can be enumerated.

The main advantage of the Enum typeclass is that we can use its types in list ranges. They also have defined successors and predecessors, which you can get with the succ and pred functions.

Types in this class: (), Bool, Char, Ordering, Int, Integer, Float and Double.

```
ghci 50> ['a' .. 'e']
"abcde"

ghci 51> ['LT' .. 'GT']
['LT','EQ','GT']
```
5.6 Bounded

Bounded members have an upper and a lower bound.

The Bounded class is used to name the upper and lower limits of a type. Ord is not a superclass of Bounded since types that are not totally ordered may also have upper and lower bounds.

The Bounded class may be derived for any enumeration type; minBound is the first constructor listed in the data declaration and maxBound is the last. Bounded may also be derived for single-constructor datatypes whose constituent types are in Bounded.

From package base class Bounded a

\begin{verbatim}
ghci 52> [3..5]
[3,4,5]
\end{verbatim}

\begin{verbatim}
ghci 53> succ 'B'
'C'
\end{verbatim}

\begin{verbatim}
ghci 54> :! hoogle -- info Bounded
Prelude class Bounded a

The Bounded class is used to name the upper and lower limits of a type. Ord is not a superclass of Bounded since types that are not totally ordered may also have upper and lower bounds.

The Bounded class may be derived for any enumeration type; minBound is the first constructor listed in the data declaration and maxBound is the last. Bounded may also be derived for single-constructor datatypes whose constituent types are in Bounded.

From package base class Bounded a

\end{verbatim}

\begin{verbatim}
ghci 55> minBound :: Int
   -9223372036854775808
\end{verbatim}

\begin{verbatim}
ghci 56> maxBound :: Int
   9223372036854775807
\end{verbatim}

\begin{verbatim}
ghci 57> minBound :: Char
   '\NUL'
\end{verbatim}

\begin{verbatim}
ghci 58> maxBound :: Char
   '\1114111'
\end{verbatim}

\begin{verbatim}
ghci 59> minBound :: Bool
   False
\end{verbatim}

\begin{verbatim}
ghci 60> maxBound :: Bool
   True
\end{verbatim}
minBound and maxBound are interesting because they have a type of \((\text{Bounded } a) \Rightarrow a\). In a sense, they are polymorphic constants.

ghci 61> :t minBound
minBound :: Bounded a \Rightarrow a

ghci 62> :t maxBound
maxBound :: Bounded a \Rightarrow a

All tuples are also part of Bounded if the components are also in it.

ghci 63> maxBound :: (Bool, Int, Char)
   (True, 9223372036854775807, ‘\1114111’)

5.7 Num

Num is a numeric typeclass. Its members have the property of being able to act like numbers.

ghci 64> :! hoogle -- info Num
Prelude class (Eq a, Show a) => Num a
   Basic numeric class.
   Minimal complete definition: all except negate or (-)
   From package base class (Eq a, Show a) => Num a

Let’s examine the type of a number.

ghci 65> :t 20
20 :: Num a \Rightarrow a

It appears that whole numbers are also polymorphic constants. They can act like any type that’s a member of the Num typeclass.

ghci 66> 20 :: Int
20

ghci 67> 20 :: Integer
20

ghci 68> 20 :: Float
20.0
Those are types that are in the `Num` typeclass. If we examine the type of `*`, we’ll see that it accepts all numbers.

It takes two numbers of the same type and returns a number of that type. That’s why `(5 :: Int) * (6 :: Integer)` will result in a type error whereas `5 * (6 :: Integer)` will work just fine and produce an `Integer` because 5 can act like an `Integer` or an `Int`.

To join `Num`, a type must already be friends with `Show` and `Eq`.

### 5.8 Integral

`Integral` is also a numeric typeclass. `Num` includes all numbers, including real numbers and integral numbers. `Integral` includes only integral (whole) numbers, i.e., `Int` and `Integer`.

`Floating` includes only floating point numbers, so `Float` and `Double`.
6 fromIntegral: converting integral numbers to general numbers

A very useful function for dealing with numbers is fromIntegral. It has a type declaration of fromIntegral :: (Num b, Integral a) ⇒ a → b.

From its type signature, we see that it takes an integral number and turns it into a more general number. That’s useful when you want integral and floating point types to work together nicely.

For instance, the length function has a type declaration of length :: [a] → Int instead of having a more general type of (Num b) ⇒ length :: [a] → b.

If we try to get the length of a list and then add it to 3.2, we’ll get an error because we tried to add together an Int and a floating point number. To get around this, we use fromIntegral.

Notice that fromIntegral has several class constraints in its type signature. That’s completely valid and as you can see, the class constraints are separated by commas inside the parentheses.