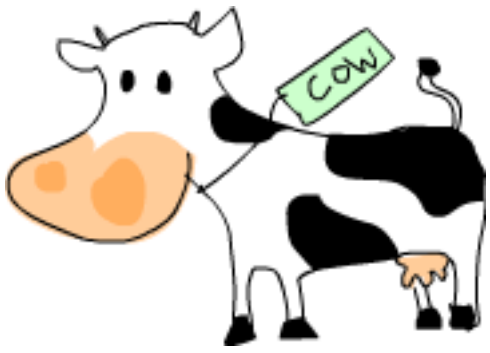
Types and Typeclasses

Believe the type



Previously 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 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.

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

Now we'll use GHCI to examine the types of some expressions. We'll do that by using the `:t` command which, followed by any valid expression, tells us its type. Let's give it a whirl.

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```

Here we see that doing `:t` on an expression prints out the expression followed by `::` and its type. `::` is read as "has type of". Explicit types are always denoted with the first letter in capital case. `'a'`, as it would seem, has a type of `Char`. It's not hard to conclude that it stands for *character*. `True` is of a `Bool` type. That makes sense. But what's this? Examining the type of `"HELLO!"` yields a `[Char]`. The square brackets denote a list. So we read that as it being *a list of characters*. Unlike lists, each tuple length has its own type. So the expression of `(True, 'a')` has a type of `(Bool, Char)`, whereas an expression such as `('a','b','c')` would have the type of `(Char, Char, Char)`. `4 == 5` will always return `False`, so its type is `Bool`.

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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 :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

`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. But 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:

```
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.

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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.

Here's an overview of some common types.

`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. Usually on 32-bit machines the maximum possible `Int` is 2147483647 and the minimum is -2147483648.

`Integer` stands for, er … also integer. The main difference is that it's not bounded so it can be used to represent really really big numbers. I mean like really big. `Int`, however, is more efficient.

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

```
ghci> factorial 50
30414093201713378043612608166064768844377641568960512000000000000
```

`Float` is a real floating point with single precision.

```
circumference :: Float -> Float
circumference r = 2 * pi * r
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
ghci> circumference 4.0
25.132742
```

`Double` is a real floating point with double the precision!

```
circumference' :: Double -> Double
circumference' r = 2 * pi * r
```

```
ghci> circumference' 4.0
25.132741228718345
```

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

`Char` represents a character. It's denoted by single quotes. A list of characters is a string.

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 `()` is also a type which can only have a single value: **`()`**

Type variables

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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

```
ghci> :t head
head :: [a] -> a
```

Hmmm! What is this `a`? Is it a type? Remember that we previously stated that types are written in capital case, so it can't exactly be a type. Because it's not in capital case it's actually a **type variable**. That means that `a` can be of any type. This is much like generics in other languages, only in Haskell it's much more powerful because it allows us to easily write very general functions if they 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> :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

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

type variables, they don't have to be different types. It just states that the first component's type and the return value's type are the same.

Typeclasses 101

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. A lot of people coming from OOP get confused by typeclasses because they think they are like classes in object oriented languages. Well, they're not. You can think of them kind of as Java interfaces, only better.

What's the type signature of the `==` function?

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

Note: the equality operator, `==` is a function. So are `+`, `*`, `-`, `/` and pretty much all operators. If a function is comprised only of special characters, it's considered an infix function by default. If we want to examine its type, pass it to another function or call it as a prefix function, we have to surround it in parentheses.

Interesting. We see a new thing here, the `=>` symbol. 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

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

`Bool`. The type of those two values must be a member of the `Eq` class (this was the class constraint).

The `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 `Eq` class. All standard Haskell types except for IO (the type for dealing with input and output) and functions are a part of the `Eq` typeclass.

The `elem` function has a type of `(Eq a) => a -> [a] -> Bool` because it uses `==` over a list to check whether some value we're looking for is in it.

Some basic typeclasses:

`Eq` is used for types that support equality testing. The functions its members implement are `==` and `/=`. So if there's an `Eq` class constraint for a type variable in a function, it uses `==` or `/=` 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> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

`Ord` is for types that have an ordering.

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

All the types we covered so far except for functions are part of **`Ord`**. **`Ord`** covers all the standard comparing functions such as **`>`**, **`<`**, **`>=`** and **`<=`**. The **`compare`** function takes two **`Ord`** members of the same type and returns an ordering. `Ordering` is a type that can be **`GT`**, **`LT`** or **`EQ`**, meaning *greater than*, *lesser than* and *equal*, respectively.

To be a member of **`Ord`**, a type must first have membership in the prestigious and exclusive **`Eq`** club.

```
ghci> "Abrakadabra" < "Zebra"
True
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

Members of `Show` can be presented as strings. All types covered so far except for functions are a 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

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

us as a string.

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

`Read` is sort of the opposite typeclass of **`Show`**. The **`read`** function takes a string and returns a type which is a member of **`Read`**.

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

So far so good. Again, all types covered so far are in this typeclass. But what happens if we try to do just **`read "4"`**?

```
ghci> read "4"
<interactive>:1:0:
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

What GHCI is telling us here is 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 knows we want some type that is part of the `Read` class, it just doesn't know which one. Let's take a look at the type signature of `read`.

```
ghci> :t read
read :: (Read a) => String -> a
```

See? 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. That's why we can use explicit **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. Observe:

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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

Most expressions are such that the compiler can infer what their type is by itself. But sometimes, the compiler doesn't know whether to return a value of type `Int` or `Float` for an expression like `read "5"`. To see what the type is, Haskell would have to actually evaluate `read "5"`. But since Haskell is a statically typed language, it has to know all the types before the code is compiled (or in the case of GHCI, evaluated). So we have to tell Haskell: "Hey, this expression should have this type, in case you don't know!".

`Enum` members are sequentially ordered types — they 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 predecesors, which you can get with the `succ` and `pred` functions. Types in this class: `()`, `Bool`, `Char`, `Ordering`, `Int`, `Integer`, `Float` and `Double`.

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ 'B'
'C'
```

`Bounded` members have an upper and a lower bound.

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False
```

`minBound` and `maxBound` are interesting because they have a type of `(Bounded a) => a`. In a sense they are polymorphic constants.

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

```
ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

`Num` is a numeric typeclass. Its members have the property of being able to act like numbers. Let's examine the type of a number.

```
ghci> :t 20
20 :: (Num t) => t
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
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.

```
ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

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`.

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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

`Floating` includes only floating point numbers, so `Float` and `Double`.

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`. I think that's there for historical reasons or something, although in my opinion, it's pretty stupid. Anyway, if we try to get a 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. So to get around this, we do `fromIntegral (length [1,2,3,4]) + 3.2` and it all works out.

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.

← Starting Out Table of contents Syntax in Functions →

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com