Haskell
Types and Typeclasses

http://igm.univ-mlv.fr/~vialette/?section=teaching

Stéphane Vialette

LIGM, Université Gustave Eiffel

November 21, 2022

Believe the type

Believe the type

One of Haskell's greatest strenghts is its powerful type system.

In Haskell, every expression's type is known at compile time, which leads to safer code.

Haskell has type inference.

Explicit type declaration

Explicit types are always denoted with the first letter in capital case

```
λ: :type True
True :: Bool
λ: :type 'a'
'a' :: Char
λ: :type "hello"
"hello" :: [Char]
λ: :type (True, 'a', "hello")
(True, 'a', "hello") :: (Bool, Char, [Char])
```

Explicit type declaration

```
λ: :type 1 == 2
1 == 2 :: Bool
λ: :type 1
1 :: Num a => a
λ: :type 1.0
1.0 :: Fractional a => a
λ: :type (1/0)
(1/0) :: Fractional a => a
```

Explicit type declaration

Functions have also types.

```
-- filter out lowercase letters
remNonUpperCase :: [Char] -> [Char]
remNonUpperCase s = [c | c <- s, c `elem` ['A'..'Z']]

-- add three integers
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Common haskell types

Int stands for integers.

Int is bounded which means that it has a minimum value and a maximum value.

Integer is also used to store integers, but it is not bounded.

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

```
λ: factorial 40
-7060926234624000
λ:
```

Common haskell types

`Int` stands for integers.

`Int` is bounded which means that it has a minimum value and a maximum value.

`Integer` is also used to store integers, but it is not bounded.

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

```
λ: :type product
product :: (Foldable t, Num a) => t a -> a
λ: factorial 40
815915283247897734345611269596115894272000000000
```

Common haskell types

Int stands for integers.

Int is bounded which means that it has a minimum value and a maximum value.

Integer is also used to store integers, but it is not bounded.

```
factorial n = product [1..n]
```

```
λ: :type factorial
factorial :: (Enum a, Num a) => a -> a
λ: factorial 40
815915283247897734345611269596115894272000000000
```

Common haskell types

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

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

```
λ: circumference 4.0
25.132742
```

Common haskell types

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

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

```
λ: circumference 4.0
25.132741228718345
```

Common haskell types

Type inference.

```
circumference :: Floating a => a -> a
circumference r = 2 * pi * r
```

```
λ: circumference 4.0
25.132741228718345
λ: :type circumference 4.0
circumference 4.0 :: Floating a => a
```

Type variables

```
λ: :type head
head :: [a] -> a
```

Because a is not in capital case it's actually a **type variable**.

That means that a can be of any type.

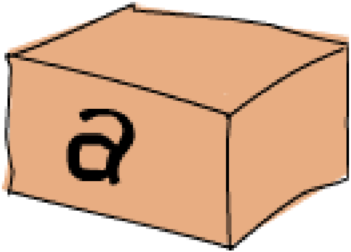
This is much like generics in other languages, only in Haskell it's much more powerful because it allows us to easily write very general functions if they don't use any specific behavior of the types in them.

Functions that have type variables are called **polymorphic functions**.

Type variables

```
λ: :type head
head :: [a] -> a
```

The type declaration of head states that it takes a list of any type and returns one element of that type.



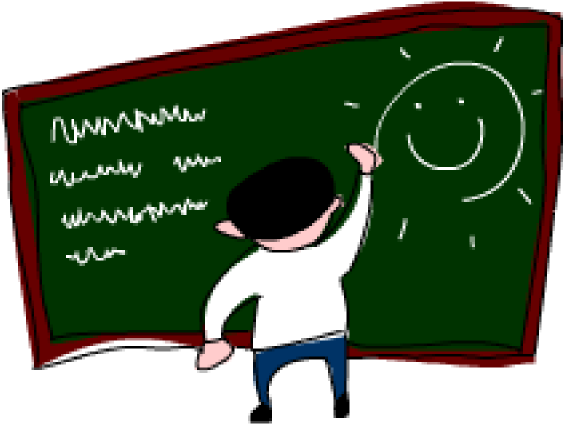
Type variables

```
λ: :type fst
fst :: (a, b) -> a
λ: :type snd
snd :: (a, b) -> b
```

Note that just because `a` and `b` are different 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



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. They're not. You can think of them kind of as Java interfaces, only better.

Typeclasses 101

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

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

Typeclasses 101

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

```
λ: :type (==)
(==) :: Eq a => a -> a -> Bool
```

Everything before the `=>` symbol is called a **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.

Typeclasses 101

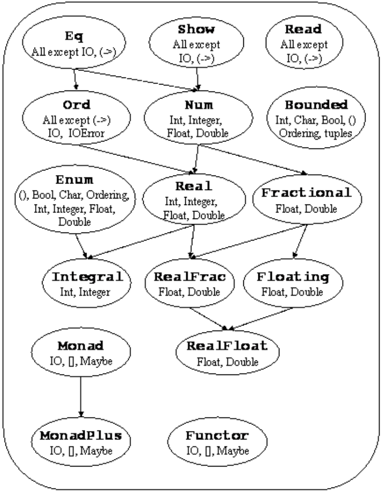
What's the type signature of the == function?

```
λ: :type (==)
(==) :: Eq a => a -> a -> Bool
```

Everything before the => symbol is called a **class constraint**.

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.

Typeclasses 101



Typeclasses 101

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.

Typeclasses 101

Basic typeclasses

`Eq` is used for types that support equality testing.

```
λ: 7 == 7
True
λ: 7 /= 7
False
λ: 'a' == 'a'
True
λ: "Hello" == "Hello"
True
λ: 3.432 == 3.432
True
```

Typeclasses 101

Basic typeclasses

`Eq` is used for types that support equality testing.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

Typeclasses 101

Basic typeclasses

`Ord` is for types that have an ordering.

```
λ: :type (>)
(>) :: 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.

Typeclasses 101

Basic typeclasses

`Ord` is for types that have an ordering.

```
λ: "Abrakadabra" < "Zebra"
True
λ: 5 >= 2
True
```

Typeclasses 101

Basic typeclasses

Ord is for types that have an ordering.

```
λ: "Abrakadabra" `compare` "Zebra"
LT
λ: 5 `compare` 3
GT
λ: 'a' `compare` 'a'
EQ
λ: :type compare
compare :: Ord a => a -> a -> Ordering
λ:
```

Ord is for types that have an ordering.

```
class (Eq a) => Ord a  where
   (<)  :: a -> a -> Bool
   (<=) :: a -> a -> Bool
   (>=) :: a -> a -> Bool
   (>)  :: a -> a -> Bool
   max  :: a -> a -> a
   min  :: a -> a -> a
```

Typeclasses 101

Basic typeclasses

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 us as a string.

Typeclasses 101

Basic typeclasses

Members of `Show` can be presented as strings.

```
λ:  :type show
show :: Show a => a -> String
λ:  show 5
"5"
λ:  show 'a'
"'a'"
λ:  show "toto"
"\"toto\""
λ:  show True
"True"
```

Typeclasses 101

Basic typeclasses

Members of `Show` can be presented as strings.

```
class Show a where
  show :: a -> String
  -- showList and showsPrec omitted
```

Typeclasses 101

Basic typeclasses

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.

```
λ: read "True" || True
True
λ: read "1" + 2
3
λ: read "1.2" * 3.4
4.08
λ: read "[1,2,3,4]" ++ [5]
[1,2,3,4,5]
```

Typeclasses 101

Basic typeclasses

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.

```
λ: read "True"
*** Exception: ghci.read: no parse
λ: read "1"
*** Exception: ghci.read: no parse
λ: read "1.2"
*** Exception: ghci.read: no parse
λ: read "[1,2,3,4]"
*** Exception: ghci.read: no parse
```

Typeclasses 101

Basic typeclasses

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.

```
λ: :type 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. 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.

Typeclasses 101

Basic typeclasses

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.

```
λ: read "True"::Bool
True
λ: read "1"::Int
1
λ: read "1.2"::Float
1.2
λ: read "[1,2,3,4]"::[Int]
[1,2,3,4]
```

Typeclasses 101

Basic typeclasses

`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`.

Typeclasses 101

Basic typeclasses

`Enum` members are sequentially ordered types – they can be enumerated.

```
λ: ['a'..'e']
"abcde"
λ: [LT .. GT]
[LT,EQ,GT]
λ: [3 .. 5]
[3,4,5]
λ: succ 'a'
'b'
λ: succ LT
EQ
λ: succ 1
2
```

Typeclasses 101

Basic typeclasses

Enum members are sequentially ordered types – they can be enumerated.

```
class Enum a where
  succ, pred     :: a -> a
  toEnum         :: Int -> a
  fromEnum       :: a -> Int
  enumFrom       :: a -> [a]                -- [n..]
  enumFromThen   :: a -> a -> [a]           -- [n,n'..]
  enumFromTo     :: a -> a -> [a]           -- [n..m]
  enumFromThenTo :: a -> a -> a -> [a]      -- [n,n'..m]
```

Typeclasses 101

Basic typeclasses

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

`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.

Typeclasses 101

Basic typeclasses

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

```
λ: maxBound :: Char
'\1114111'
λ: maxBound :: Bool
True
λ: minBound :: Bool
False
λ: maxBound :: (Bool, Int, Char)
(True,9223372036854775807,'\1114111')
```

Typeclasses 101

Basic typeclasses

Bounded members have an upper and a lower bound.

```
class  Bounded a  where
  minBound :: a
  maxBound :: a
```

Typeclasses 101

Basic typeclasses

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

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

```
λ: :type 5
5 :: Num a => a
```

Typeclasses 101

Basic typeclasses

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

```
λ: 20 :: Int
20
λ: 20 :: Integer
20
λ: 20 :: Float
20.0
λ: 20 :: Double
20.0
```

Typeclasses 101

Basic typeclasses

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

```
λ: :type (*)
(*) :: 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`.

`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`.

```
class (Real a, Enum a) => Integral a  where
  quot, rem, div, mod :: a -> a -> a
  quotRem, divMod     :: a -> a -> (a,a)
  toInteger           :: a -> Integer
```

Typeclasses 101

Basic typeclasses

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.

Typeclasses 101

Basic typeclasses

A very useful function for dealing with numbers is `fromIntegral`.

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.

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.