Typeclasses and Polymorphism

Stephen A. Edwards

Columbia University

Fall 2022
Polymorphism and Type Variables

Typeclasses
- Show and other derived type classes

Parameterized Types: Maybe

The type keyword

The Either Type

Lists as an Algebraic Data Type
- Defining Your Own Infix Operators

Specifying and Implementing Type Classes

The Functor Type Class

Kinds: The Type of Types

Numeric Conversions
Polymorphism and Type Variables

Haskell has excellent support for polymorphic functions.

Haskell supports *parametric polymorphism*, where a value may be of *any* type.

Haskell also supports *ad hoc polymorphism*, where a value may be one of a *set of types* that support a particular group of operations.

**Parametric polymorphism: the `head` function**

Prelude> :t head
head :: [a] -> a

Here, `a` is a *type variable* that ranges over *every possible type*.

**Prelude> :t fst**

fst :: (a, b) -> a

Here, `a` and `b` are distinct type variables, which may be *equal* or *different*.
Ad Hoc Polymorphism and Type Classes

Haskell’s ad hoc polymorphism is provided by **Type Classes**, which specify a group of operations that can be performed on a type (think Java Interfaces).

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

“The (==) function takes two arguments of type `a`, which must be of the `Eq` class, and returns a `Bool`”

Members of the `Eq` class can be compared for equality

A type may be in multiple classes; multiple types may implement a class
# Common Typeclasses

**Eq**  
Equality: `==` and `/=`

**Ord**  
Ordered: `Eq` and `>`, `>=`, `<`, `<=`, `max`, `min`, and `compare`, which gives an `Ordering`: `LT`, `EQ`, or `GT`

**Enum**  
Enumerable: `succ`, `pred`, `fromEnum`, and `toEnum` (conversion to/from `Int`), and list ranges

**Bounded**  
`minBound`, `maxBound`

**Num**  
Numeric: `+`, `-`, `*`, `negate`, `abs`, `signum`, and `fromInteger`

**Real**  
`Num`, `Ord`, and `toRational`

**Integral**  
`Real`, `Enum`, and `quot`, `rem`, `div`, `mod`, `toInteger`, `quotRem`, `divMod`

**Show**  
Can be turned into a string: `show`, `showList`, and `showsPrec` (operator precedence)

**Read**  
Opposite of `Show`: string can be turned into a value: `read` et al.
Ord, Enum, and Bounded Typeclasses

Prelude> :t (>)
(>) :: Ord a => a -> a -> Bool

Prelude> :t compare
compare :: Ord a => a -> a -> Ordering

Prelude> :t succ
succ :: Enum a => a -> a

Prelude> maxBound :: Int
9223372036854775807

Prelude> minBound :: Char
'\NUL'

Prelude> maxBound :: Char
'\1114111'

Prelude> minBound :: (Char, Char)
('\NUL','\NUL')
The Num Typeclass

Prelude> :t 42
42 :: Num p => p  -- Numeric literals are polymorphic
Prelude> :t (+)
(+) :: Num a => a -> a -> a  -- Arithmetic operators are, too

Prelude> :t 1 + 2
1 + 2 :: Num a => a
Prelude> :t (1 + 2) :: Int
(1 + 2) :: Int :: Int  -- Forcing the result type
Prelude> :t (1 :: Int) + 2
(1 :: Int) + 2 :: Int  -- Type of one argument forces the type

Prelude> :t (1 :: Int) + (2 :: Double)
<interactive>:1:15: error:
  * Couldn't match expected type 'Int' with actual type 'Double'
  * In the second argument of '(+)', namely '(2 :: Double)'
    In the expression: (1 :: Int) + (2 :: Double)
The Integral and Fractional Typeclasses

Prelude> :t div
div :: Integral a => a -> a -> a  -- div is integer division
Prelude> :t toInteger
toInteger :: Integral a => a -> Integer  -- E.g., Int to Integer
Prelude> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b  -- Make more general
Prelude> 1 + 3.2
4.2  -- Fractional
Prelude> (1 :: Int) + 3.2
   * No instance for (Fractional Int) arising from the literal '3.2'
   * In the second argument of '(+)', namely '3.2'
      In the expression: (1 :: Int) + 3.2
      In an equation for 'it': it = (1 :: Int) + 3.2
Prelude> fromIntegral (1 :: Integer) + 3.2
4.2  -- Num + Fractional
Prelude> :t (/)
(/) :: Fractional a => a -> a -> a  -- Non–integer division
The Show Typeclass

Show is helpful for debugging

Prelude> :t show
show :: Show a => a -> String

Prelude> show 3
"3"

Prelude> show 3.14159
"3.14159"

Prelude> show pi
"3.141592653589793"

Prelude> show True
"True"

Prelude> show (True, 3.14)
"(True,3.14)"

Prelude> show ["he","llo"]
"["he","llo"]"
Printing User-Defined Types: Deriving Show

*Main> Circle 10 20 30

<interactive>:9:1: error:
  ∗ No instance for (Show Shape) arising from a use of 'print'
  ∗ In a stmt of an interactive GHCi command: print it

Add deriving (Show) to make the compiler generate a default show:

data Shape = Circle Float Float Float | Rectangle Float Float Float Float
  deriving Show

*Main> Circle 10 20 30
  Circle 10.0 20.0 30.0
*Main> show $ Circle 10 20 30
  "Circle 10.0 20.0 30.0"
Many Automatic Derivations

```
data Bool = False | True            -- Standard Prelude definition
deriving (Eq, Ord, Enum, Read, Show, Bounded)
```

```
Prelude> True == True
True                       -- Eq
Prelude> False < False
False                      -- Ord
Prelude> succ False
True                       -- Enum
Prelude> succ True
Prelude> read "True" :: Bool
True                       -- Read
Prelude> show False
"False"                    -- Show
Prelude> minBound :: Bool
False                      -- Bounded
```
Parameterized Types: Maybe

A safe replacement for null pointers

```haskell
data Maybe a = Nothing | Just a
```

The *Maybe* type constructor is a function with a type parameter (*a*) that returns a type (*Maybe a*).
Maybe In Action

Useful when a function may “fail” and you don’t want to throw an exception

Prelude> :m + Data.List
Prelude Data.List> :t uncons
uncons :: [a] -> Maybe (a, [a])
Prelude Data.List> uncons [1,2,3]
Just (1,[2,3])
Prelude Data.List> uncons []
Nothing

Prelude Data.List> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
Prelude Data.List> lookup 5 [(1,2),(5,10)]
Just 10
Prelude Data.List> lookup 6 [(1,2),(5,10)]
Nothing
Data.Map: Multiple Type Parameters

Prelude Data.Map> :k Map
Map :: * -> * -> *

Prelude Data.Map> :t empty
empty :: Map k a

Prelude Data.Map> :t singleton (1::Int) "one"
singleton (1::Int) "one" :: Map Int [Char]

Note: while you can add type class constraints to type constructors, e.g.,

data Ord k => Map k v = ... 

it’s bad form to do so. By convention, to reduce verbosity, only functions that actually rely on the type classes are given such constraints.
The type Keyword: Introduce an Alias

Prelude> type AssocList k v = [(k, v)]
Prelude> :k AssocList
AssocList :: * -> * -> *
Prelude> :

Prelude| lookup :: Eq k => k -> AssocList k v -> Maybe v
Prelude| lookup _ [] = Nothing
Prelude| lookup k ((x,v):xs) | x == k = Just v
Prelude| | otherwise = lookup k xs
Prelude| :}
Prelude> :t lookup
lookup :: Eq k => k -> AssocList k v -> Maybe v
Prelude> lookup 2 [(1,"one"), (2,"two")]
Just "two"
Prelude> lookup 0 [(1,"one"), (2,"two")]
Nothing

Prelude> :t [(1,"one"), (2,"two")]
[(1,"one"), (2,"two")]: Num a => [(a, [Char])]
data Either a b = Left a | Right b
    deriving (Eq, Ord, Read, Show)

Prelude> :k Either
Either :: * -> * -> *
Prelude> Right 20
Right 20
Prelude> Left "Stephen"
Left "Stephen"
Prelude> :t Right "Stephen"
Right "Stephen" :: Either a [Char]  -- Only second type inferred
Prelude> :t Left True
Left True :: Either Bool b
Prelude> :k Either Bool
Either Bool :: * -> *

Either: Funky Type Constructor Fun
Either: Often a more verbose Maybe

By convention, Left = “failure,” Right = “success”

Prelude> type AssocList k v = [(k,v)]
Prelude> :{
Prelude| lookup :: String -> AssocList String a -> Either String a
Prelude| lookup k [] = Left $ "Could not find " ++ k
Prelude| lookup k ((x,v):xs) | x == k = Right v
Prelude| | otherwise = lookup k xs
Prelude| :}
Prelude> lookup "Stephen" [(("Douglas",42),("Don",0))]
Left "Could not find Stephen"
Prelude> lookup "Douglas" [(("Douglas",42),("Don",0))]
Right 42
**data List a = Cons a (List a)  -- A recursive type**

| Nil

**deriving (Eq, Ord, Show, Read)**

*Main> :t Nil
Nil :: List a  -- Nil is polymorphic

*Main> :t Cons
Cons :: a -> List a -> List a  -- Cons is polymorphic

*Main> :k List
List :: * -> *

*Main> Nil
Nil

*Main> 5 `Cons` Nil
Cons 5 Nil

*Main> 4 `Cons` (5 `Cons` Nil)
Cons 4 (Cons 5 Nil)

*Main> :t 'a' `Cons` Nil
'a' `Cons` Nil :: List Char  -- Proper type inferred
Lists of Our Own with User-Defined Operators

```
infixr 5 :.
data List a = a :. List a
    | Nil
    deriving (Eq, Ord, Show, Read)
```

Haskell symbols are  ! # $ % & * + . / < = > ? @ \ ^ | - ~

A (user-defined) operator is a symbol followed by zero or more symbols or :

A (user-defined) constructor is a : followed by one or more symbols or :

```
*Main> (1 :. 2 :. 3 :. Nil) :: List Int
1 :. (2 :. (3 :. Nil))
*Main> :t (:.)
(:.) :: a -> List a -> List a
```
Fixity of Standard Prelude Operators

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>infixr</td>
</tr>
<tr>
<td>8</td>
<td>infixr</td>
</tr>
<tr>
<td>7</td>
<td>infixl</td>
</tr>
<tr>
<td>6</td>
<td>infixl</td>
</tr>
<tr>
<td>5</td>
<td>infixr</td>
</tr>
<tr>
<td>4</td>
<td>infixr</td>
</tr>
<tr>
<td>3</td>
<td>infixr</td>
</tr>
<tr>
<td>2</td>
<td>infixr</td>
</tr>
<tr>
<td>1</td>
<td>infixl</td>
</tr>
<tr>
<td>0</td>
<td>infixr</td>
</tr>
</tbody>
</table>

-- Highest precedence
-- Right-associative
-- Left-associative
-- : is the only builtin
-- Non-associative
-- Lowest precedence

*Main> (1::Int) == 2 == 3
<interactive>:9:1: error:
   Precedence parsing error
      cannot mix '==' [infix 4] and '==' [infix 4] in the
      same infix expression
The List Concatenation Operator

```
infixr 5 ++.          -- Define operator precedence & associativity
(++)                  :: List a -> List a -> List a
Nil ++. ys = ys
(x :: xs) ++. ys = x :: (xs ++. ys)
```

```
*Main> (1 :: 2 :: 3 :: Nil ++. 4 :: 5 :: Nil) :: List Int
1 :: (2 :: (3 :: (4 :: (5 :: Nil))))
```

The only thing special about lists in Haskell is the [], syntax

```
*Main> :k List
List :: * -> *
*Main> :k []
[] :: * -> *
```

Our `List` type constructor has the same kind as the built-in list constructor `[]`
data Tree a = Node a (Tree a) (Tree a)  -- Unbalanced binary tree
    | Nil
    deriving (Eq, Show, Read)

singleton :: a -> Tree a
singleton x = Node x Nil Nil

insert :: Ord a => a -> Tree a -> Tree a
insert x Nil = singleton x
insert x n@(Node a left right) = case compare x a of
    LT -> Node a (insert x left) right
    GT -> Node a left (insert x right)
    EQ -> n

fromList :: Ord a => [a] -> Tree a
fromList = foldr insert Nil

toList :: Tree a -> [a]
toList Nil = []
toList (Node a l r) = toList l ++ [a] ++ toList r
member :: Ord a => a -> Tree a -> Bool
member _ Nil = False
member x (Node a left right) = case compare x a of
    LT -> member x left
    GT -> member x right
    EQ -> True

*Main> t = fromList ([8,6,4,1,7,3,5] :: [Int])
*Main> t
Node 5 (Node 3 (Node 1 Nil Nil) (Node 4 Nil Nil))
    (Node 7 (Node 6 Nil Nil) (Node 8 Nil Nil))
*Main> toList t
[1,3,4,5,6,7,8]
*Main> 1 `member` t
True
*Main> 42 `member` t
False
class Eq a where  -- Standard Prelude definition of Eq
  (==), (/=) :: a -> a -> Bool  -- The class: names & signatures
  x /= y  = not (x == y)  -- Default implementations
  x == y  = not (x /= y)

data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
  Red  == Red    = True  -- Suffices to only supply
  Green == Green = True  -- an implementation of ==
  Yellow == Yellow = True
  _      == _     = False  -- "deriving Eq" would have been easier
Implementing Show

\[
\text{instance \ Show \ TrafficLight \ where}
\]
\[
\text{show \ Red \ = \ "Red \ Light"}
\]
\[
\text{show \ Green \ = \ "Green \ Light"}
\]
\[
\text{show \ Yellow \ = \ "Yellow \ Light"}
\]

*Main> \text{show Yellow}
"Yellow \ Light"

*Main> \text{[Red, Yellow, Green]}
[Red \ Light, Yellow \ Light, Green \ Light] \quad \text{-- GHCi uses show}

*Main> \text{:k Maybe}
Maybe :: \star \rightarrow \star \quad \text{-- A polymorphic type constructor}

*Main> \text{:k Eq}
Eq :: \star \rightarrow \text{Constraint} \quad \text{-- Like a polymorphic type constructor}

*Main> \text{:k Eq TrafficLight}
Eq \text{ TrafficLight} :: \text{Constraint} \quad \text{-- Give it a type to make it happy}
The MINIMAL Pragma: Controlling Compiler Warnings

```haskell
infix 4 ==., /=.

class MyEq a where
  {−# MINIMAL (==.) | (/=.)#−}
  (==.), (/=.) :: a -> a -> Bool
  x /=. y = not (x ==. y)
  x ==. y = not (x /=. y)

instance MyEq Int where

instance MyEq Integer where
  x ==. y = (x `compare` y) == EQ
```

Prelude> :load myeq
[1 of 1] Compiling Main

myeq.hs:9:10: warning:
[-Wmissing-methods]
  * No explicit implementation for either '==.' or '/=.'
  * In the instance declaration for 'MyEq Int'
    instance MyEq Int where
      x ==. y = (x `compare` y) == EQ
```

The MINIMAL pragma tells the compiler what to check for. Operators are , (and) and | (or). Parentheses are allowed.
Eq (Maybe t)

data Maybe t = Just t | Nothing

instance Eq t => Eq (Maybe t) where
    Just x == Just y = x == y  -- This comparison requires Eq t
    Nothing == Nothing = True
    _ == _ = False

The Standard Prelude includes this by just deriving Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
{-# MINIMAL (==) | (=/) #-}
instance [safe] Eq TrafficLight
instance (Eq a, Eq b) => Eq (Either a b)
instance Eq a => Eq (Maybe a)
instance Eq a => Eq [a]
instance Eq Ordering
instance Eq Int
instance Eq Float
instance Eq Double
instance Eq Char
instance Eq Bool
instance (Eq a, Eq b) => Eq (a, b)
instance (Eq a, Eq b, Eq c) => Eq (a, b, c)
instance (Eq a, Eq b, Eq c, Eq d) => Eq (a, b, c, d)
class ToBool a where
  toBool :: a -> Bool

instance ToBool Bool where
  toBool = id
  -- Identity function

instance ToBool Int where
  toBool 0 = False
  -- C-like semantics
  toBool _ = True

instance ToBool [a] where
  toBool [] = False
  -- JavaScript, python semantics
  toBool _ = True

instance ToBool (Maybe a) where
  toBool (Just _) = True
  toBool Nothing = False
Now We Can toBool Bools, Ints, Lists, and Maybes

*Main> :t toBool
toInt :: ToBool a => a -> Bool

*Main> toBool True
True

*Main> toBool (1 :: Int)
True

*Main> toBool "dumb"
True

*Main> toBool []
False

*Main> toBool [False]
True

*Main> toBool $ Just False
True

*Main> toBool Nothing
False
The Functor Type Class: Should be “Mappable”†

\[
\text{class Functor } f \text{ where}
\]
\[
fmap :: (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b
\]
\[
(<$) :: b \rightarrow f \ a \rightarrow f \ b
\]
\[
m <$ b = fmap (\_ \rightarrow b)
\]

If \( f :: a \rightarrow b \),
\[
bs = fmap f \ as
\]

applies \( f \) to every \( a \) in \( as \) to give \( bs \); \( bs \)
\[
=\ as <$ x replaces every \( a \) in \( as \) with \( x \).
\]

Here, \( f \) is a type constructor that takes an argument, like Maybe or List

Prelude> :k Functor
Functor :: (* -> *) -> Constraint

† “Functor” is from Category Theory
Functor Instances for $\ast \rightarrow \ast$ Kinds

data [] a = [] | a : [a] -- The List type: not legal syntax

instance Functor [] where -- Prelude definition
  fmap = map -- The canonical example

data Maybe t = Nothing | Just t -- Prelude definition

instance Functor Maybe where
  fmap _ Nothing = Nothing -- No object a here
  fmap f (Just a) = Just (f a) -- Apply f to the object in Just a

data Tree a = Node a (Tree a) (Tree a) | Nil -- Our binary tree

instance Functor Tree where
  fmap f Nil = Nil
  fmap f (Node a lt rt) = Node (f a) (fmap f lt) (fmap f rt)
Functor Either a

```haskell
data Either a b = Left a | Right b
```

instance Either does not type check because Either :: * -> * -> *

The Prelude definition of `fmap` only modifies `Right`

```haskell
instance Functor (Either a) where
    fmap _ (Left x) = Left x
    fmap f (Right y) = Right (f y)
```

This works because Either a :: * -> * has the right kind
Kinds: The Types of Types

Prelude> :k Int
Int :: * -- A concrete type

Prelude> :k [Int]
[Int] :: * -- A specific type of list: also concrete

Prelude> :k []
[] :: * -> * -- The list type constructor takes a parameter

Prelude> :k Maybe
Maybe :: * -> * -- Maybe also takes a type as a parameter

Prelude> :k Maybe Int
Maybe Int :: * -- Specifying the parameter makes it concrete

Prelude> :k Either
Either :: * -> * -> * -- Either takes two type parameters

Prelude> :k Either String
Either String :: * -> * -- Partially applying Either is OK

Prelude> :k (,)
(,) :: * -> * -> * -- The pair (tuple) constructor takes two
Type class `Tofu` expects a single type argument `t`

`j` must take an argument `a` and produce a concrete type, so `j :: * -> *`

`t` must take arguments `a` and `j`, so `t :: * -> (* -> *) -> *`

Let’s invent a type constructor of kind `* -> (* -> *) -> *`. It has to take two type arguments; the second needs to be a function of one argument.

```haskell
data What a b = What (b a) deriving Show
```

```haskell
Prelude> :k What
What :: * -> (* -> *) -> *  -- Success
```
What?

```
data What a b = What (b a) deriving Show
```

```
Prelude> :t What "Hello"
What "Hello" :: What Char []
Prelude> :t What (Just "Ever")
What (Just "Ever") :: What [Char] Maybe
```

*What* holds any type that is a “parameterized container,” what *Tofu* wants:

```
Prelude> :k What
What :: * -> (* -> *) -> *
Prelude> :k Tofu
Tofu :: (* -> (* -> *) -> *) -> Constraint
Prelude> instance Tofu What where tofu x = What x
Prelude> tofu (Just 'a') :: What Char Maybe
What (Just 'a')
Prelude> tofu "Hello" :: What Char []
What "Hello"
```
Prelude> data Barry t k a = Barry a (t k)
Prelude> :k Barry
Barry :: (\* -> \*) -> \* -> \* -> \*  -- Bizarre kind, by design
Prelude> :t Barry (5::Int) "Hello"
Barry (5::Int) "Hello" :: Barry [] Char Int

A Barry is two objects: any type and one built from a type constructor

Prelude> :k Functor
Functor :: (\* -> \*) -> Constraint  -- Takes a one-arg constructor

instance Functor (Barry t k) where  -- Partially applying Barry
fmap f (Barry x y) = Barry (f x) y  -- Applying f to first object

Prelude> fmap (+1) (Barry 5 "Hello")
Barry 6 "Hello"  -- It works!
Prelude> fmap show (Barry 42 "Hello")
Barry "42" "Hello"
Prelude> :t fmap show (Barry 42 "Hello")
fmap show (Barry 42 "Hello") :: Barry [] Char String
class Eq a
  where
  (==), (/=) :: a -> a -> Bool

class Eq a => Ord a
  where
  compare :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max :: a -> a -> a

class Num a
  where
  (+), (-), (*) :: a -> a -> a
  negate, abs, signum :: a -> a
  fromInteger :: Integer -> a

class (Num a, Ord a) => Real a
  where
  toRational :: a -> Rational

class Enum a
  where
  succ, pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  ...
Integral Typeclasses and Conversion

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

instance Integral Int
instance Integral Word
instance Integral Integral Integer

Conversion among Integrals:

fromIntegral :: (Integral a, Num b) => a -> b
fromIntegral = fromInteger . toInteger
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a

class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate, round, ceiling, floor :: Integral b => a -> b

Conversions among Reals and Fractionals:

realToFrac :: (Real a, Fractional b) => a -> b
realToFrac = fromRational . toRational

instance RealFrac Float
instance RealFrac Double

type Rational = GHC.Real.Ratio Integer
Conversion Examples

Prelude> :t 42
42 :: Num p => p
Prelude> :t 42.0
42.0 :: Fractional p => p

Prelude> (fromIntegral (42 :: Int)) :: Word
42
Prelude> (realToFrac (42 :: Int)) :: Double
42.0
Prelude> (realToFrac (42.5 :: Float)) :: Double
42.5
Prelude> (floor (42.5 :: Double)) :: Int
42

https://wiki.haskell.org/Converting_numbers