Typeclasses

Introduction

We've mentioned typeclasses in previous chapters, but a solid understanding of them is crucial to understanding Haskell. Typeclasses abstract patterns and provide methods for working with typed data. In the Types chapter, we looked at the way typeclasses interact with type variables and numeric types. This chapter explains some important predefined typeclasses as well as going into more detail about how typeclasses work more generally. In this chapter we:

- examine the typeclasses Eq, Num, Ord, Enum, and Show;
- learn about type-defaulting typeclasses and typeclass inheritance;
- learn how to build our own typeclass and use it

What are typeclasses?

A typeclass is a way to group types that support the same operations. If a type is a part of a typeclass, it means that it supports and implements the behavior the typeclass describes.

Built-In typeclass Eq, Num, Ord, Enum, Show, …

Querying typeclasses information from REPL

To obtain information about a typeclass we run the command `:info` or `:i` follow by the typeclass name. The output shows a lot of information.

Eq typeclass

Eq typeclass is defined for types that can be equated.
When we query information about the typeclass Eq (by running the command `:info Eq`) we obtain a bunch of information about the typeclass.

1. The first part of the output tells us how the Eq typeclass is defined.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
```

```
      -- Defined in 'GHC.Classes'
```

The syntax,

- uses the keyword ***class*** and ***where***.
- declares two functions:
  - `(==) :: a -> a -> Bool` and
  - `(/=) :: a -> a -> Bool`

  the behaviors the Eq typeclass defines.
- specify the minimal implementation of the behaviors: `{-# MINIMAL (==) | (/=) #-}` . It means that, to implement the typeclass Eq you must redefine the function (==) or (/=), otherwise the default implementation will be applied.

The second part lists all the types that are part of the Eq typeclass.

```
instance Eq a => Eq [a] -- Defined in 'GHC.Classes'
instance Eq Word -- Defined in 'GHC.Classes'
instance Eq Ordering -- Defined in 'GHC.Classes'
instance Eq Int -- Defined in 'GHC.Classes'
instance Eq Float -- Defined in 'GHC.Classes'
instance Eq Double -- Defined in 'GHC.Classes'
instance Eq Char -- Defined in 'GHC.Classes'
instance Eq Bool -- Defined in 'GHC.Classes'
```

`instance Eq Bool -- Defined in 'GHC.Classes'` tells us that the type Bool implements the typeclass Eq (or is part of typeclass Eq) and it definition is in the `GHC.Classes` module. `instance Eq a => Eq [a] -- Defined in 'GHC.Classes'` tells us that the list ([]) type of object of type *a* is part of Eq typeclass if the type *a* is part of Eq typeclass and it definition is in the `GHC.Classes` module.

Other typeclasses

By running the above command (:info) as previously with Num, Ord, Enum or Show we obtain à bunch of information about the typeclass.

Typeclasses and Types

conversely by querying the information about à type, it lists all typeclasses the type is part of. Example : Querying the type of Bool we obtain information made of two parts.
The first part of the output tells us how the Bool type is defined: `data Bool = False | True -- Defined in 'GHC.Types'`

The second part lists all the typeclasses the Bool type is part of. We can read that Bool type implement Eq, Ord, Show, Read, Enum and Bounded typeclasses. As shown below in the second part

```
instance Eq Bool -- Defined in 'GHC.Classes'
instance Ord Bool -- Defined in 'GHC.Classes'
instance Show Bool -- Defined in 'GHC.Show'
instance Read Bool -- Defined in 'GHC.Read'
instance Enum Bool -- Defined in 'GHC.Enum'
instance Bounded Bool -- Defined in 'GHC.Enum'
```

Make a type part of a typeclass

In Haskell, there are two ways to make a type to be part of typeclass.

Using the deriving keyword

The syntax using the keyword deriving is used during the definition of the type. Usually when we want to use default implementations of operations. For example:
if we define

```
data Person = Person {
     fullName :: String
   , placeOfBirth :: String
   , yearOfBirth :: Int
   , height :: Float
   , phoneNumber :: String
}
let takam = Person {
   fullName = "Takam Fridolin",
   placeOfBirth = "Yaounde",
   yearOfBirth = 1925,
   height = 1.78,
   phoneNumber = "691179855"
}
```

Person is not an instance of Show typeclass. If we run in the REPL the command `takam` we will obtain the following error:

```
<interactive>:4:1: error:
    • No instance for (Show Person) arising from a use of 'print'
    • In a stmt of an interactive GHCi command: print it
```

The typeclass Show is to group things that can be printed on the screen. In the definition of Person we can make it instance of typeclass Show like this:

```
data Person = Person {
     fullName :: String
   , placeOfBirth :: String
   , yearOfBirth :: Int
   , height :: Float
   , phoneNumber :: String
} deriving (Show)
```

To make Person type part of the typeclass Ord, we could write it definition like this:

```
data Person = Person {
     fullName :: String
   , placeOfBirth :: String
   , yearOfBirth :: Int
   , height :: Float
   , phoneNumber :: String
} deriving (Show, Ord)
```

Internally, Haskell uses comparison of string, float and Integer. It compares corresponding fields.

Class and instances (Using the instance keyword)

We use the syntax:

```
instance TypeclassName TypeName where
      [implementation of the behaviors]
```

Example:

```
data Person = Person {
     fullName :: String
   , placeOfBirth :: String
   , yearOfBirth :: Int
   , height :: Float
   , phoneNumber :: String
}
instance Eq Person where
   (==) p1 p2 = (height p1) == (height p2)
```

```haskell
data DayOfWeek = Mon | Tue | Weds | Thu | Fri | Sat | Sun
data Date = Date DayOfWeek Int
instance Eq DayOfWeek where
  (==) Mon Mon = True
  (==) Tue Tue = True
  (==) Weds Weds = True
  (==) Thu Thu = True
  (==) Fri Fri = True
  (==) Sat Sat = True
  (==) Sun Sun = True
  (==) _ _ = False
```

Let's consider our type Person defined as follow:

```haskell
data Person = Person {
      fullName :: String
    , placeOfBirth :: String
    , yearOfBirth :: Int
    , height :: Float
    , phoneNumber :: String
}
```

to make the type Person a part of the typeclass Ord, using instance keyword, we first query information about the Ord typeclass by running the command :info Ord

```
:info Ord
```

we obtain:

```haskell
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  {-# MINIMAL compare | (<=) #-}
      -- Defined in 'GHC.Classes'
instance Ord a => Ord [a] -- Defined in 'GHC.Classes'
instance Ord Word -- Defined in 'GHC.Classes'
instance Ord Ordering -- Defined in 'GHC.Classes'
instance Ord Int -- Defined in 'GHC.Classes'
instance Ord Float -- Defined in 'GHC.Classes'
instance Ord Double -- Defined in 'GHC.Classes'
instance Ord Char -- Defined in 'GHC.Classes'
instance Ord Bool -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g, Ord h,
          Ord i, Ord j, Ord k, Ord l, Ord m, Ord n, Ord o) =>
         Ord (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g, Ord h,
          Ord i, Ord j, Ord k, Ord l, Ord m, Ord n) =>
         Ord (a, b, c, d, e, f, g, h, i, j, k, l, m, n)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g, Ord h,
          Ord i, Ord j, Ord k, Ord l, Ord m) =>
         Ord (a, b, c, d, e, f, g, h, i, j, k, l, m)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g, Ord h,
          Ord i, Ord j, Ord k, Ord l) =>
         Ord (a, b, c, d, e, f, g, h, i, j, k, l)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g, Ord h,
          Ord i, Ord j, Ord k) =>
         Ord (a, b, c, d, e, f, g, h, i, j, k)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g, Ord h,
          Ord i, Ord j) =>
         Ord (a, b, c, d, e, f, g, h, i, j)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g, Ord h,
          Ord i) =>
         Ord (a, b, c, d, e, f, g, h, i)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g,
          Ord h) =>
         Ord (a, b, c, d, e, f, g, h)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g) =>
         Ord (a, b, c, d, e, f, g)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f) =>
         Ord (a, b, c, d, e, f)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d, Ord e) => Ord (a, b, c, d, e)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c, Ord d) => Ord (a, b, c, d)
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b, Ord c) => Ord (a, b, c)
```

```
  -- Defined in 'GHC.Classes'
instance (Ord a, Ord b) => Ord (a, b) -- Defined in 'GHC.Classes'
instance Ord () -- Defined in 'GHC.Classes'
instance (Ord a, Ord b) => Ord (Either a b)
  -- Defined in 'Data.Either'
instance Ord a => Ord (Maybe a) -- Defined in 'GHC.Maybe'
instance Ord Integer
  -- Defined in 'integer-gmp-1.0.2.0:GHC.Integer.Type'
```

From this information, it is enough to implement compare or (<=) function.
We choose to implement the (<=) function. This is how we proceed.

```
instance Eq Person where
    (==) p1 p2 = (height p1) == (height p2)
```

```
instance Ord Person where
    (<=) p1 p2 = (height p1) <= (height p2)
```

After loading the code in the REPL, if we query the information about the typeclass Ord the custom type Person will appear in the list of types that implement Ord typeclass. Now it is rightful to write in our code p1 < p2 or p1 <= p2, …

Custom typeclasses

To create our own typeclass we use the syntax:

```
class [(MotherClass a) =>] MyCustomTypeclass a where
        op1 :: ....
        op2 :: ....
        op3 :: ....
        ...
```

MyCustomTypeclass is the new typeclass we are creating. a represents the type that will implement the type class. op1, op2, op3, … are the the operation that the typeclass exposes.

Example:

```
class Numberish a where
   fromNumber :: Integer -> a
   toNumber :: a -> Integer
```

```
newtype Age = Age Integer deriving (Eq, Show)
```

```
instance Numberish Age where
   fromNumber n = Age n
   toNumber (Age n) = n
```

```
newtype Year = Year Integer deriving (Eq, Show)
instance Numberish Year where
   fromNumber n = Year n
   toNumber (Year n) = n
```

Class inheritance

Classes can inherit from other classes. For example, here is the main part of the definition of Ord in Prelude:

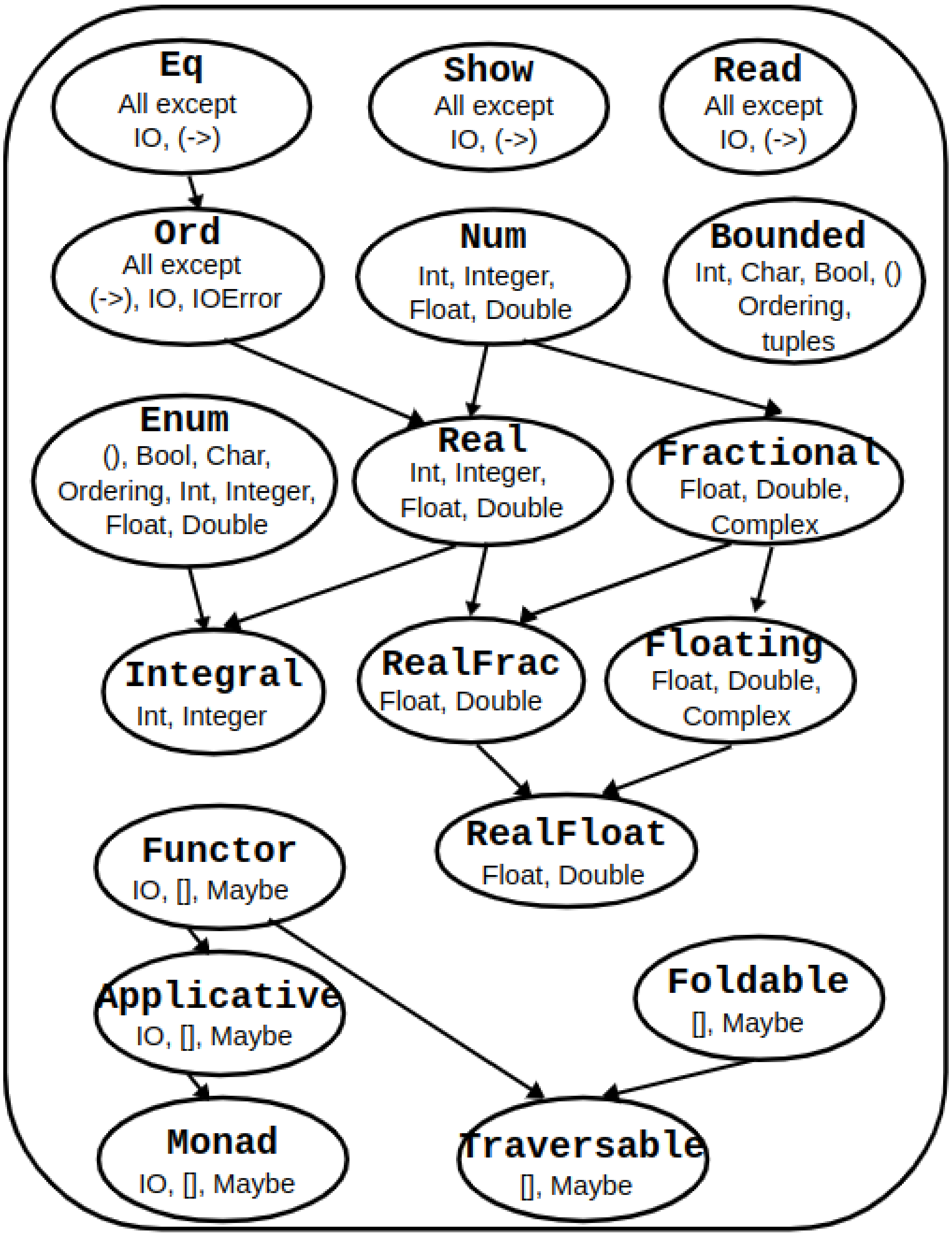
```
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a
```

The point here is that Ord inherits from Eq. This is indicated by the => notation in the first line, which mirrors the way classes appear in type signatures. Here, it means that for a type to be an instance of Ord it must also be an instance of Eq, and hence needs to implement the == and /= operations.

A class can inherit from several other classes: just put all of its superclasses in the parentheses before the =>. Let us illustrate that with yet another Prelude quote:

```
class  (Num a, Ord a) => Real a  where
    -- | the rational equivalent of its real argument with full
precision
    toRational          ::  a -> Rational
```

Classes hierarchy



This diagram, adapted from the Haskell Report, shows the relationships between the classes and types in the Standard Prelude. The names in bold are the classes, while the non-bold text stands for the types that are instances of each class ((->) refers to functions and [], to lists). The arrows linking classes indicate the inheritance relationships, pointing to the inheriting class.

Type constraints

With all pieces in place, we can go full circle by returning to the very first example involving classes in this course:

```
(+) :: (Num a) => a -> a -> a
```

Here, the arguments `x` and `y` must be of the same type, and that type must be an instance of `Num class`. Furthermore, the final argument `t` must be of some (possibly different) type that is also an instance of `Num`. This example also displays clearly how constraints propagate from the functions used in a definition (in this case, `(+)` and `Num`) to the function being defined.

Note:

Beyond simple type signatures, type constraints can be introduced in a number of other places:

- instance declarations (typical with parameterized types);
- class declarations (constraints can be introduced in the method signatures in the usual way for any type variable other than the one defining the class);
- data declarations, where they act as constraints for the constructor signatures.

Note

Type constraints in data declarations are less useful than it might seem at first. Consider:

```
data (Num a) => Foo a = F1 a | F2 a String
```

Here, Foo is a type with two constructors, both taking an argument of a type a which must be in Num. However, the (Num a) => constraint is only effective for the F1 and F2 constructors, and not for other functions involving Foo. Therefore, in the following example…

```
fooSquared :: (Num a) => Foo a -> Foo a
fooSquared (F1 x)   = F1 (x * x)
fooSquared (F2 x s) = F2 (x * x) s
```

... even though the constructors ensure a will be some type in Num we can't avoid duplicating the constraint in the signature of fooSquared.

Example

To provide a better view of the interplay between types, classes, and constraints, we will present a very simple and somewhat contrived example. We will define a Located class, a

Movable class which inherits from it, and a function with a Movable constraint implemented using the methods of the parent class, i.e. Located.

```haskell
-- Location, in two dimensions.
class Located a where
    getLocation :: a -> (Int, Int)

class (Located a) => Movable a where
    setLocation :: (Int, Int) -> a -> a

-- An example type, with accompanying instances.
data NamedPoint = NamedPoint
    { pointName :: String
    , pointX    :: Int
    , pointY    :: Int
    } deriving (Show)

instance Located NamedPoint where
    getLocation p = (pointX p, pointY p)

instance Movable NamedPoint where
    setLocation (x, y) p = p { pointX = x, pointY = y }

-- Moves a value of a Movable type by the specified displacement.
-- This works for any movable, including NamedPoint.
move :: (Movable a) => (Int, Int) -> a -> a
move (dx, dy) p = setLocation (x + dx, y + dy) p
    where
    (x, y) = getLocation p
```

Advice:

Do not read too much into the `Movable` example just above; it is merely a demonstration of class-related language features. It would be a mistake to think that every single functionality which might be conceivably generalized, such as `setLocation`, needs a type class of its own. In particular, if all your `Located` instances should be able to be moved as well then `Movable` is unnecessary - and if there is just one instance there is no need for type classes at all! Classes are best used when there are several types instantiating it (or if you expect others to write additional instances) and you do not want users to know or care about the differences between the types. An extreme example would be `Show`: general-purpose functionality implemented by an immense number of types, about which you do not need to know a thing before calling `show`. In the following chapters, we will explore a number of important type classes in the libraries; they provide good examples of the sort of functionality which fits comfortably into a class.