User-Define Types and Type Classes 2

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Instance of the `Eq` Type Class

- Let's look at the information about the `Eq` type class:

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

- According to the above information:
  - There are two functions == and /=
  - At **minimum**, an instance of the `Eq` type class must implement either == or /=
- Suppose we are implementing a new data type named `TrafficLight`

```
data TrafficLight = Red | Yellow | Green
```

- We are going to make the `TrafficLight` an instance of the `Eq` type class without using automatic derive

Manual Deriving (`Eq`)

- To make our `TrafficLight` type be an instance of the `Eq` type class, we need to at least implement either == or /= function
- We need to use the `instance` keyword followed by function definitions:

```
instance Eq TrafficLight where
    Red == Red = True
    Yellow == Yellow = True
    Green == Green = True
    _ == _ = False
```

- Note that the definition of the `Eq` type class starts with

```
class Eq a where
   :
```

- Since `a` is a type variable, we need to declare an instance by

```
instance Eq TrafficLight where
   :
```

- Note that /= function can be derived from == since

```
x == y = not (x /= y)
x /= y = not (x == y)
```

Deriving the Show Type Class Manually

- If we want to create our own string representation, we need to make our type an instance to the `Show` type class
- Let's take a look at its information:

```
ghci> :info Show
class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
  {-# MINIMAL showsPrec | show #-}
  :
```

- At the minmum, we need to implement the `show` function:

```
instance Show TrafficLight where
    show Red = "Red Light"
    show Yellow = "Yellow Light"
    show Green = "Green Light"
```

- Now, we can do the following:

```
ghci> Red
Red Light
ghci> [Red, Yellow, Green]
[Red Light,Yellow Light,Green Light]
```

Deriving the `Ord` Type Class Manually

- What about the `Ord` type class?
- Again, let's check its information:

```
ghci> :info Ord
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 | (<=) #-}
  :
```

- This time, it has a type constrint
- To be an instance of the `Ord` type class, the class must be an instance of the `Eq` type class first
- At the minimum, we need to impelemnt either the `compare` or $<=$ function

Deriving the `Ord` Type Class Manually

- Our `TrafficLight` is already an instance of the `Eq` type class
- Here is an implementation:

```
instance Ord TrafficLight where
    compare Red Red = EQ
    compare Red _ = LT
    compare Yellow Red = GT
    compare Yellow Yellow = EQ
    compare Yellow Green = LT
    compare Green Green = EQ
    compare Green _ = GT
```

- Now we can compare values of `TrafficLight`

```
ghci> Red > Green
False
ghci> map (<Green) [Red, Yellow, Green]
[True,True,False]
```

Deriving Types with Parameters

- Recall our `BTree` type that has a type parameter:

```
data BTree a = EmptyBTree
             | BTree a (BTree a) (BTree a)
```

- If we want to make it an instance of the `Eq` type class:

```
instance Eq (BTree a) where
   :
```

- `a` is a type variable in the above example
- If comparing two binary trees require to compare data of type `m`, we have to make sure that `a` is an instance of the `Eq` class
- We need to put a constraint by delaring the following:

```
instance Eq m => Eq (BTree m) where
   :
```

The `Functor` Type Class

- Simply speaking, a functor is a thing that can be **mapped**
- One obvious example is a list
- Let's look at the information about the `Functor` type class

```
ghci> :i Functor
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  -# MINIMAL fmap #-
  :
```

- An instance of the `Functor` type class must implement the `fmap` function

The `Functor` Type Class

- Look closely at the type signature of the `fmap` function:

```
ghci> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

- The constraint `Functor f` indicates that `f` is a type variable
- Obviously, `a` and `b` are type variables
- But what are `f a` and `f b`?
  - `f a` and `f b` must be a type (`f a -> f b`)
  - This tells us that `f` is a type constructor
- Simply put `fmap` takes a function of type `a -> b` and a value of a type `f a` and returns a value of type `f b`
- Confuse?

The `Functor` Type Class

- Look closely at the type signature of the `map` function:

```
ghci> :t map
map :: (a -> b) -> [a] -> [b]
```

- In Haskell, `[ ]` of list is a type constructor
  - `[Int]` is a concrete type
  - `[Char]` is a concrete type
- The `map` function takes a function of type `a` to `b`, and a list of type `a` and returns a list of type `b`
- Think about `[ ]` as `f` in the `fmap` function signature

The `Functor` Type Class

- Recall our own list data type:

```
data MyList a = EmptyList
              | Cons a (MyList a)
              deriving (Show)
```

- If we want to define our own `myMap` function on `MyList`, it should look be

```
myMap f EmptyList = EmptyList
myMap f (Cons x xs) = Cons (f x) (myMap f xs)
```

- Let's look at the type of `myMap` compared to `fmap`:

```
ghci> :t myMap
myMap :: (t -> a) -> MyList t -> MyList a
ghci> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

- Replacing `f` my `MyList` and ignore the `Functor f`, they are practically identical
- Again, a functor is a thing that can be mapped
- Usually, a collection of values including no value

Maybe As a Functor

- To make a type an instance of the `Functor` type class, use the `instance` keyword and define the `fmap` function
- This is a part of the `Maybe`:

```
instance Functor Maybe where
    fmap f Nothing = Nothong
    fmap f (Just x) = Just (f x)
```

- Here are some examples:

```
ghci> fmap (++ "I am inside Just") (Just " **** ")
Just " **** I am inside Just"
ghci> fmap (++ "I am inside Just") Nothing
Nothing
ghci> fmap (*2) (Just 12)
Just 24
```

`Either a` as a `Functor`

- Recall the type signature of the `fmap` function:

```
ghci> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

- The type constructor `f` only takes on type as the argument
- But `Either` takes two arguments

```
data Either a b = Left a
                | Right b
                deriving (Eq, Ord, Read, Show)
```

- But `Either a` where `a` is a type, is a type constructor that takes only one argument
- This is now `Either` was defined to be an instance of `Functor`:

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

- We cannot map the `Left` since it can have different type

Kinds

- Type constructors take other types as arguments and eventually produce a **concrete type**
- A concrete type is a type that does not take any type parameter
- Example:
  - `Char` is a concrete type
  - `Maybe` is a type constructor
  - `Maybe Char` is a concrete type
  - `Nothing` and `Just` are value constructors
- Example:
  - `Int` is a concrete type
  - `[ ]` is a type constructor
  - `[Int]` is a concrete type
  - `[]` and `:` are value constructors
- Types have their labels called `kinds` (type of a type)
- To view the kind of a concrete type or type constructor, use the command `:k`

```
ghci> :k Int
Int :: *
```

Kind

- The $\star$ means a **concrete type**
- Let's check the type `Maybe`

```
ghci> :k Maybe
Maybe :: * -> *
```

- It says that `Maybe` takes one concrete type and returns a concrete type

```
ghci> :k Maybe Char
Maybe Char :: *
ghci> :k Maybe [Int]
Maybe [Int] :: *
```

- Let's check the kind of the `Either`:

```
ghci> :k Either
Either :: * -> * -> *
ghci> :k Either [Char]
Either [Char] :: * -> *
ghci> :k Either Int Char
Either Int Char :: *
```

Kinds

- We can also check the kind of `[ ]`

```
ghci> :k [ ]
[ ] :: * -> *
ghci> :k [Char]
[Char] :: *
ghci> :k [Int]
[Int] :: *
ghci> :k [[Char]]
[[Char]] :: *
```

- We use the `:k` command to get the kind of a type
- We use the `:t` command to get the type of a value
- Remember the information about the `Functor` type class?

```
ghci> :i Functor
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  -# MINIMAL fmap #-
  :
```

- The signature of `f` indicates that `f` must be a type constructor of kind `* -> *`