Haskell Tutorial: Typeclasses

November 7, 2019

[1]:
```
:opt no-lint
```

1 Typeclasses

Typeclasses describe a set of types that have a common interface and behaviour. We have already seen typeclasses such as `Ord` or `Num` or `Show`. In a nutshell, a typeclass defines common behaviour of types that belong to the typeclass. A type belonging to a typeclass implements the functions and behaviour defined by the typeclass. Note that the notion of a typeclass is very different from the concept of classes in object-oriented programming languages like C++ or Java. The closest concepts in popular object-oriented programming languages are probably Java interfaces.

The perhaps simplest standard typeclass is `Eq`. Types that are *instances* of the `Eq` typeclass implement an operator (a function) that can be used to determine whether two values of the type are equal. Note that the way equality is determined is type specific. For numeric types, equality is defined by comparing two numerical values. For lists, equality is defined by comparing all elements of two lists. There are also types where equality is not defined for all values or where it would be infeasible to decide equality of two values. The `Eq` typeclass can be defined in Haskell as follows:

[2]:
```
class Eq a where
    (==) :: a -> a -> Bool
    (/=)  :: a -> a -> Bool
```

The definition says that type `a` is an instance of the class `Eq` if there are (overloaded) operations `==` and `/=`, of the appropriate type, defined on it. A type implementing the `Eq` typeclass, i.e., a type that is an instance of the `Eq` typeclass, must implement these operation with the given type signature and the associated semantics. The functions (operations) of a typeclass are also called *methods* of the typeclass. We can go even further and provide default implementations in case this is possible. Note that in this particular example, the default implementations refer to each other, which means that a type instance of `Eq` only needs to define one of the two methods.

[3]:
```
class Eq a where
    (==) :: a -> a -> Bool
    (==) a b = not (a /= b) -- default implementation
    (/=)  :: a -> a -> Bool
    (/=) a b = not (a == b) -- default implementation
```

Lets define a simple enumeration type for weekdays. We can make our `Weekday` type an instance

of the `Eq` typeclass by defining the methods of the `Eq` typeclass. Since we have default implementations for the methods, it is sufficient to define only one of the methods. The other method will then be available immediately due to its default implementation.

[4]:
```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun

instance Eq Weekday where
  Mon == Mon = True
  Tue == Tue = True
  Wed == Wed = True
  Thu == Thu = True
  Fri == Fri = True
  Sat == Sat = True
  Sun == Sun = True
  _   == _   = False

Sun == Sun
Sun /= Sat
```

```
True
```

```
True
```

The next example shows how we can define lists of a given type to be an instance of `Eq`. We assume that the type of the list elements already is an instance of `Eq`.

[5]:
```
instance Eq a => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _      == _      = False

[Sun, Mon, Tue] == [Sun, Mon, Tue]
```

```
True
```

When we write generic (polymorphic) code, we often refer to types that belong to certain type classes, i.e., to all types that have a certain common behaviour. We can express that a type `a` should be an instance of a certain typeclass by expressing a constraint on a type, which we call a *context*. The expression `Eq a` says that we constrain the type a to those types that are instances of the `Eq` typeclass.

The following example shows how our `Eq` typeclass can be used. In order to determine whether a certain element is in a list, we test whether the first element is equal to the element we are looking for or we test whether the element we are looking for is in the tail of the list. Our definition of the `elem` function can be used with all list element types that are instances of the `Eq` typeclass. This is expressed using the `Eq a` context in the type signature of our function. Since numeric types are instances of the `Eq` typeclass, we can determine whether a number is an element of a list of

numbers. Similarly, since characters are instances of the `Eq` typeclass, we can determine whether a character is an element of a list of characters (a string).

[1]:
```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem y (x:xs) = y == x || elem y xs
```

It is possible to extend typeclasses. The `Ord` typeclass defines order relations. The `Ord` typeclass can be defined as an extension of the `Eq` typeclass. We say that `Eq` is a *superclass* of `Ord` or that `Ord` is a *subclass* of `Eq`. The definition below also provides default method implementations.

[6]:
```
data Ordering = LT | EQ | GT

instance Eq Ordering where
    LT == LT = True
    EQ == EQ = True
    GT == GT = True
    _  == _  = False

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

    compare x y | x == y    = EQ  -- default implementation
                | x <= y     = LT
                | otherwise = GT

    x <= y = compare x y /= GT  -- default implementation
    x < y = compare x y == LT  -- default implementation
    x >= y = compare x y /= LT  -- default implementation
    x > y = compare x y == GT  -- default implementation

    max x y | x <= y     =  y  -- default implementation
            | otherwise =  x
    min x y | x <= y     =  x  -- default implementation
            | otherwise =  y
```

An interesting typeclass is the `Functor` typeclass. It generalizes the `map` function we already know from lists to any parametric types that can contain values. The Functor typeclass is defined as follows:

[7]:
```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Note that the type variable `f` is applied to other types in `f a` and `f b`.

[8]:
```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving (Show)

instance Functor Tree where
    fmap f (Leaf x)     = Leaf (f x)
    fmap f (Branch l r) = Branch (fmap f l) (fmap f r)

t = Branch (Leaf "hello") (Leaf "world")
print t
print $ fmap reverse t
```

```
Branch (Leaf "hello") (Leaf "world")


Branch (Leaf "olleh") (Leaf "dlrow")
```

The type `Maybe` is an instance of the `Functor` typeclass as is the standard list type. Hence we can apply functions to `Maybe` values in a list or `Maybe` values in a tree; the `Functor` typeclass nicely hides the differences between a list and tree.

[15]:
```
-- instance Functor Maybe where
--    fmap f Nothing  = Nothing
--    fmap f (Just x) = Just (f x)

-- instance Functor [] where
--    fmap = map

print $ fmap (*2) (Just 1)

l = [Just 1, Just 2, Just 3, Nothing]
t = Branch (Branch (Leaf (Just 1)) (Leaf (Just 2))) (Leaf Nothing)

print $ fmap (fmap (*2)) l
print $ fmap (fmap (*2)) t
```

```
Overlapping instances for Functor Maybe arising from a use of `fmap'
Matching instances:
  instance Functor Maybe -- Defined at <interactive>:1:10
  instance Functor Maybe -- Defined at <interactive>:1:10
In the second argument of `($)', namely `fmap (* 2) (Just 1)'
In the expression: print $ fmap (* 2) (Just 1)
In an equation for `it': it = print $ fmap (* 2) (Just 1)
```

[10]:
```

```