More on type classes and instances

In this section, we study in detail two important type classes:

1. The type class `Ord` for ordered types.
2. The type class `Num` for numeric types.

We also study an example of an *instance declaration with constraints*: `instance Ord a => Ord [a]`. This instance tells us that whenever `a` is an instance of `Ord`, then so is `[a]`.

The type `Ordering` and the typeclass `Ord`

There is also a video on this section.

The type class `Ord` implements the idea that elements of a type can be **compared not only for equality, but also for less/greater than**. A comparison on a type `a` is a map `compare : a -> a -> Ordering`, where the type `Ordering` is defined as follows:

```
data  Ordering  =  LT | EQ | GT
          deriving (Eq, Ord, Enum, Read, Show, Bounded)
```

Here, `LT` stands for *less than*, `EQ` stands for *equal*, and `GT` stands for *greater than*.

The following defines the `Ord` class for types `a` which are in the `Eq` class.

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

        -- Minimal complete definition:
        --      (<=) or compare
        -- Using compare can be more efficient for complex types.
    compare x y
         | x == y    =  EQ
         | x <= y    =  LT
         | otherwise =  GT

    x <= y           =  compare x y /= GT
    x <  y           =  compare x y == LT
    x >= y           =  compare x y /= LT
    x >  y           =  compare x y == GT

-- note that (min x y, max x y) = (x,y) or (y,x)
    max x y
         | x <= y    =  y
         | otherwise =  x
    min x y
         | x <= y    =  x
         | otherwise =  y
```

There is a seeming circularity between the definition of `Ordering` and `Ord`, in that each one refers to the other. We may see this as a mutually recursive definition.

For example, the `compare` function on lists over `a` (where `a` is already equipped with a function `compare`) is implemented as follows:

```
instance (Ord a) => Ord [a] where
  compare [] []         = EQ
  compare [] (_:_)      = LT
  compare (_:_) []      = GT
  compare (x:xs) (y:ys) = case compare x y of
```

```
                              EQ    -> compare xs ys
                              other -> other
```

Exercises

1. From reading the code, explain how two lists are compared.
2. Run some examples of comparisons between lists to confirm or refute your explanation.

This video gives an explanation of the implementation of `compare :: [a] -> [a] -> Ordering`.

The type class `Num`

Consider the following examples

```
Prelude> 5 + 3
8
Prelude> 3.14159 + 2.71828
5.85987
```

Here, the operator `+` acts on any type that is an instance of the `Num` typeclass: for any such type `a`, the function `(+)` is of type `a -> a -> a`, used as an infix operator.

We can use the `:info` command to find out what operations the `Num` typeclass provides:

```
Prelude> :info Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
        -- Defined in ‘GHC.Num’
instance Num Word -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’
```

This shows that any type `a` that is an instance of `Num` comes with operations `(+)`, `(-)`, `(*)`,..., `fromInteger`. (In particular, `1` without a type annotation is not an integer, but instead an element of any type `a` that is an instance of `Num`; specifically, the image of the integer `1 :: Integer` under the function `fromInteger :: Integer -> a`.)

We also see that there are five **instances** of the type class `Num` defined, for the types `Word`, `Integer`, `Int`, `Float`, and `Double`. When we want to consider `1` as an element of a particular numeric type, we can do that by **annotating** it with that type, e.g.,

```
Prelude> :type 1 :: Word
1 :: Word :: Word
```

Here, we **check** that `1 :: Word` instead of asking `ghci` to **infer** the type of `1` for us. `ghci` then just needs to check that `Word` is an instance of `Num`. Similarly, for `(+)` we can check

```
Prelude> :type (+) :: Integer -> Integer -> Integer
(+) :: Integer -> Integer -> Integer
  :: Integer -> Integer -> Integer
```

This check fails for types for which no instance of `Num` has been declared, e.g., for `Char`:

```
Prelude> :type (+) :: Char -> Char -> Char

<interactive>:1:1: error:
    • No instance for (Num Char) arising from a use of ‘+’
    • In the expression: (+) :: Char -> Char -> Char
```

Quiz time

Test your understanding by taking this quiz. Don't worry, it is not marked, and you can take it as many times as you want.

See also

3. A blog post comparing Java and Haskell: https://mmhaskell.com/blog/2019/1/28/why-haskell-iv-typeclasses-vs-inheritance. Do you agree with it? Why (not)?

Summary

1. We have studied in detail one function that uses both pattern matching on lists and a `case` expression (a more general form of pattern matching).
2. We have seen how instances can be derived automatically from other instances, using the example `instance Ord a => Ord [a]`.
3. We have taken a look at the type class `Num` for number types.
4. We have seen how to use a type annotation to force a Haskell expression to have a particular type, e.g., `1 :: Word`.