Fixed unresolved git conflict
Gabriel Huddy authored 1 day ago

71a29576

typeclasses.md 6.49 KB

Type classes and instances

- See the prelude for the current version of the language for all predefined classes and their instances.

Introduction

Video: This introduction is also available as a recording.

We frequently want to check if two values in Haskell are equal.

Observation: We only want to compare two values **of the same type** for equality. It does not make any sense to ask whether, e.g., a Boolean is equal to a character.

We would hence be tempted to write a **polymorphic function**

```
(==) : a -> a -> Bool
```

However, it is not always possible to decide if two values of a given type are equal.

Exercise: Find a type `a` whose values cannot be compared for equality with a Boolean function `a -> a -> Bool`.

(One such type is given in the video.)

The solution is given by **type classes**:

1. A **type class** is an interface for a set of operations on one (or more) types.
2. An **instance** of a type class is any type for which we have implemented the interface.

In Haskell, the operation `(==)` has the following type:

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

Here,

1. `Eq` is a **type class**, and
2. for any type `a` that is an **instance** of the type class `Eq`, `(==)` is a function of type `a -> a -> Bool` - but only for such `a`.
3. `Eq a` in `Eq a => a -> a -> Bool` is a **class constraint**:

Exercise

1. Run, and understand, the following examples:
    1. `False == 'c'`
    2. `False == True`
    3. `False == not`
    4. `False == not True`
    5. `not == id`
    6. `[not] == [ (id :: Bool -> Bool) ]`

Explanation: See the video.

For the example 6., Haskell understands that it should look for an instance of `Eq [Bool -> Bool]`. Since there is a generic instance `Eq a => Eq [a]`, Haskell proceeds to look for an instance of `Eq (Bool -> Bool)`. Alas, there is no such instance, as we know from example 5.

2. Explain, in your own words, why `(++)` does not require any class constraints, but `(==)` does.

The type class `Eq`

Video: See here.

We obtain information about the type class `Eq` as follows (some text is removed for legibility):

```
Prelude> :info Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
        -- Defined in ‘GHC.Classes’
instance (Eq a, Eq b) => Eq (Either a b)
  -- Defined in ‘Data.Either’
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 a, Eq b, Eq c) => Eq (a, b, c)
  -- Defined in ‘GHC.Classes’
instance (Eq a, Eq b) => Eq (a, b) -- Defined in ‘GHC.Classes’
instance Eq () -- Defined in ‘GHC.Classes’
instance Eq Integer
  -- Defined in ‘integer-gmp-1.0.2.0:GHC.Integer.Type’
instance Eq a => Eq (Maybe a) -- Defined in ‘GHC.Base’
```

This tells us the following:

1. The type class `Eq` provides two functions, `(==)` and `(\=)`.
2. To implement the type class on a type `a`, we have to implement at least one of `(==) :: a -> a -> Bool` or `(\=) :: a -> a -> Bool`.
3. Many instances of `Eq` are implemented, e.g., for `Word`, `Ordering`, `Int`,... Furthermore, we have **derived** instances:
    1. If types `a` and `b` are instances, then the type `(a,b)` of pairs of values in `a` and `b` is also an instance.
    2. Similarly, if `a` is an instance of `Eq`, so is the type `[a]` of lists of values in `a`.

Further information that is **not** printed above:

1. When only one of `(==)` and `(\=)` is provided by the user when implementing the instance, the other is defined automatically as its **negation**, e.g., `x \= y = not (x == y)`.
2. The information does not say what the implementation of `(==)` is for any of the instances. This requires looking at the source code.

Summary: type classes and instances

- A `class` in Haskell is like an interface in Java.
- We implement a class in Haskell using the keyword `instance`.
- Only types that are introduced using `data` or `newtype` can be made instances of classes (although GHC has some extensions to get around this...).
- It is only possible to declare a **single** instance of a class for any given `data` type or `newtype` (although GHC has some extensions to get around this...).
- In a function type, everything before `=>` is a **class constraint**: the function is only available for such types that are an instance of the mentioned classes.

Inheritance: Extending a type class

Just like a Java interface can extend an interface, a type class can extend a type class. Consider the following example:

```
Prelude> :i Ord
class Eq a => Ord a where
...
```

Here, the type class `Ord` (which we look at in detail below) extends the type class `Eq`. In other words, in order to turn a type `a` into an instance of `Ord`, we first need to turn it into an instance of `Eq`. This will be studied in more detail later.

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.

Exercises:

1. Find all the basic instances of the type class `Bounded` that are defined in the GHC Prelude (the libraries that are loaded when starting `ghci`, without importing any additional libraries). Find out what `minBound` and `maxBound` are for each of the instances.
2. What type classes do the type classes `Fractional`, `Floating`, `Integral` extend? What functions do they provide? Which type class would you choose to implement a trigonometric calculus?
3. Another type class:
    1. Which type class defines the function `enumFromTo`?
    2. Evaluate `enumFromTo` on elements of each instance of that type class.
    3. Explain the different output between `:type enumFromTo 4 8` and `:type enumFromTo 4 (8 :: Int)`. If you are unsure about the answer, ask in the Teams chat.
4. Why does Haskell only allow **one** instance of any type class on a given type?