Week 2
Type classes in Haskell

CS4012
Topics in Functional Programming
Michaelmas Term 2020
Glenn Strong <glenn.Strong@scss.tcd.ie>
Type classes

The type of equality

- We test for equality in Haskell using the infix (==) operator

```haskell
GHCi> 1 == 2
False
GHCi> [1,2,3] == (reverse [3,2,1])
True
```

- What is the type of this (==) function?
Type classes

The type of equality

• It compares things of the same type to give a Boolean result. Perhaps some polymorphic type like:

\[(==) :: a \rightarrow a \rightarrow \text{Bool}\]

• What does GHCi think?

\[
\text{GHCi} > :t (==) \\
(==) :: (\text{Eq } a) \Rightarrow a \rightarrow a \rightarrow \text{Bool}
\]

• What’s that thing?
Type classes

Ad-Hoc Polymorphism

• Equality is “polymorphic” all right

\[ (==) :: a \rightarrow a \rightarrow \text{Bool} \]

• However it’s polymorphism is *ad-hoc*.

• We need a different specific implementation for each type.

• Basically, \((==)\) is not one function. It’s a *collection* of functions, indexed by type.
Type classes

Ad-Hoc Polymorphism

• Contrast with *parametric* polymorphism.

• For example, the “length” function:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

• Length only relies on properties of the list type. The type of the list *contents* don’t matter.

• So the same function works for all lists.
Type classes

Ad-Hoc Polymorphism

• Ad-hoc polymorphism is very common in programming languages

• Using a single symbol (say “+”) to denote lots of different (but related) operators is often called “overloading”.

• In many languages this overloading is built in and permanently fixed

• In Haskell it is provided through a language feature called “type classes”, so we can roll our own functions with this kind of overloading.
Type classes

Defining type classes in Haskell

In order to define our own potential name/operator overloading we:

• Need to specify the name/operator involved (e.g. ==)
• Need to describe its pattern of use (e.g. a -> a -> Bool)
• Need to name the “class” for the concept (e.g. Eq)
Type classes

Defining type instances in Haskell

Then, to give a specific actual operator, for a given type (e.g. Bool) we:

• Give an implementation of (==) for that specific type

In Haskell this is called giving an *instance* of a class.
Type classes

Example: defining the Equality class

We could define the class `Eq` like this:

```haskell
class Eq a where
  (==) :: a -> a -> Bool
```

This introduces the name “Eq” for the class of operations over types, and lists the operations of that class (== is the only one in this example).

Any specific type (like “Char”, say) can be a member of the class `Eq`. We just have to provide a suitable implementation of “==”. 
Type classes

Example: Giving an instance of the equality class

We can provide an instance of Eq for booleans like this:

```haskell
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

As usual “_” is a wildcard pattern matching anything.
Type classes

Example: Giving an instance of the equality class

We can define other instances for any other types for which equality tests are desired.

In the case of Eq, many of the tests are just some build-in test, but when we define new types we can provide Eq instances for them this way.

The Haskell Prelude contains definitions for all the usual types, of course.
Type classes

Example: classes with more than one member

Of course, if you have (==) then you should also have some not-equal operation:

```haskell
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

This says:

- An instance has two functions, (==) and (/=)
  - They have the same type, so I used a bit of Haskell shorthand to avoid repeating the type signature
- The definition of (/=) is in terms of (==), so there’s no need to make someone provide it.
Type classes

Example: the real Equality class

The real Eq class is a tiny bit more complex...

class Eq a where
  (==), (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)

What witchcraft is this?

• The defaults are *circular* definitions!

• When creating an instance you can provide an overriding definition of *either* (==) or (/=), whichever is easier to write. The other function can then use the default implementation.

• You could even override both (perhaps for efficiency reasons)
Type classes

Class resolution

How does Haskell handle resolving class operation use?

Consider this (well typed) expression:

```haskell
x == 3 && y == False
```

So `x` has type `Int` and `y` has type `Bool`

The compiler:

• Sees the `(==)` operator, and notes it belongs to the `Eq` class

• seeing that `3 :: Int` it concludes the first use of `(==)` is of type `Int -> Int -> Bool`, and so it generates code for that expression using the `Int` instance.

• A similar analysis for the second `(==)` generates Boolean equality code.
Type classes

Class resolution

Now consider this expression, assuming that there is a user-defined type MyType with MyCons as a constructor:

\[ x == 3 \land y == \text{False} \lor z == \text{MyCons} \]

Also assume that we have *not* declared an instance of Eq for this type.

Seeing the third use of (==) the compiler looks for an instance for MyType for the Eq class, and fails to find one.

We get a compile-time error something like this:

No instance for (Eq MyType)
arising from a use of `==` at ...
Possible fix: add an instance declaration for (Eq MyType)

Note the helpful suggestion!
Type classes

Standard classes

Haskell has a large family of classes provided as standard.

- Relations: Eq, Ord
- Enumeration: Enum, Bounded
- Numeric: Num, Real, Integral, Fractional, Floating,…
- Textual: Show, Read
- Categorical: Applicative, Functor, Monad
Thank you

glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/