Ad hoc polymorphism via Typeclasses

A reminder about parametric polymorphism

Parametric polymorphism refers to functions with some inputs or outputs that could be of **any** type

- We can also have parametric polymorphic **types**, but that is not the focus of today

```
id :: a -> a
length :: [a] -> Int
foldr :: (a -> b -> b) -> b -> [a] -> b
```

But is this polymorphic?

Is the function

```
(+) :: a -> a -> a
```

polymorphic?

Surely not! There are many types for which this does not make sense.

And yet, it does work for more than *one* type: `Double`, `Int`, `Integer`, `Rational`, and many more.

Ad hoc polymorphism

`(+)` is an example of **ad hoc** polymorphism.

It is defined for certain types, namely though those that belong to a certain **typeclass** (called `Num`).

`(+)` has a different definition for e.g. `Int` vs `Double`.

The act of giving a function name different definitions for different types is called **overloading**.

Overloading

A symbol is **overloaded** if it has two (or more) meanings, distinguished by type.

There is completely different code written in the Prelude to define `(+)` for `Int` and for Double

In Haskell overloading is resolved at compile time.

If a function is overloaded, then the compiler must choose between the possible algorithms– this process is called overload resolution.

Ad hoc versus Parametric Polymorphism

- Parametric polymorphic functions use **one algorithm** to operate on arguments of any types
- Ad hoc polymorphism (may) use a **different algorithm for different types**

Overloading is an important language feature because **many useful functions are not parametric polymorphic**: they work only for types that support certain operations.

```
sum :: Num a => [a] -> a
```

`sum` only works for types that have a `(+)` and `0` defined.

Typeclasses

- Typeclasses are a language mechanism in Haskell designed to support general overloading in a principled way.
- They allow users to express which overloaded operations they expect to be able to use on their data, and to define their own overloaded operations.

Components of the Haskell typeclass mechanism

- Typeclass declaration
- Typeclass `instance` declaration
  - or `deriving`
- Context

Typeclass Declaration

defines a set of operations and their types and gives the set a name

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

Any type `a` that belongs to the `Eq` typeclass has two operators == and `/=` that return a `Bool` when applied to two values of that type.

Therefore if you want your type a to be an instance of the class Eq then you have to write your own code for == and `/=`

Instance Declaration

When you use the instance keyword, you then must define all the required functions for that typeclass.

```
instance Eq Bool where

  b == c = case b of
    True -> c
    False -> not c

  b /= c = not (b == c)
```

Default Method

Some classes offer default behaviour for functions, which will be used unless it is overwritten.

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

So it is actually only necessary to define one of == and /= for your instance.

Minimal Definition

Because of default behaviour, Haskell documentation tends to state what the minimal definition necessary to define an instance is

e.g. in the Prelude:

```
class Eq a where
…
Minimal complete definition
(==) | (/=)
```

This means that you can defined Eq through (==) *or* (/=) alone

Deriving

In fact `instance Eq Bool` is not defined as in the previous slides, but rather something more like

```
data Bool = False | True
  deriving Eq
```

We have already seen this many times in this course!

Can derive multiple typeclasses by e.g. `deriving (Eq,Show)`

deriving

`deriving` only works with six Prelude typeclasses

- `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, `Read`
- Significant restrictions on use with `Enum` and `Bounded`

It works only with `data` definitions built up with products and sums (not functions!) and types that are already part of the typeclass, and assumes any type variables will also be part of the same typeclass.

It gives a default definition for your typeclass functions, which may or may not be what you want.

deriving

```
data Maybe a = Nothing | Just a
  deriving Eq
```

- `Maybe a` will be an instance of Eq only if the type instantiating `a` is.
- The default behaviour will be something like:

```
m == n  = case (m,n) of
  (Nothing, Nothing) -> True
  (Just x, Just y)   -> (x == y)
  _                  -> False
```

Context

The **context** of a function's type expresses a **constraint** on polymorphism, i.e. which group of overloaded definitions we assume will exist.

```
elem :: Eq a => a -> [a] -> Bool
```

`elem` will work for any type which is an instance of `Eq`.

We can have multiple constraints in a context:

```
elem :: (Eq a, Foldable t) => a -> t a -> Bool
```

Class Extensions (Class Inclusions)

The class `Ord` **inherits** all of the operations in `Eq`,
but in addition has a set of comparison operations and minimum and maximum functions, e.g.:

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

`Eq` is a **superclass** of `Ord`.

`Ord` is a **subclass** of `Eq`.

Benefits of class inclusions

- shorter contexts: a type expression for a function that uses operations from both the `Eq` and `Ord` classes can use the context (Ord a), rather than (`Eq a, Ord a`), since `Ord` "implies" `Eq`

- functions for subclass operations can assume the existence of default functions for superclass operations.

Multiple Inheritance of Classes

classes may have more than one superclass

For example, the declaration

```
class (Eq a, Show a) => C a where ...
```

creates a class C which inherits operations from both Eq and Show.

Typeclasses of the Prelude

- **Eq**
- **Ord**
- **Enum**
- **Bounded**
- **Num**
- Real
- Integral
- Fractional
- Floating
- RealFrac
- RealFloat

- Semigroup
- Monoid
- Functor
- Applicative
- Monad
- MonadFail
- Foldable
- Traversable
- **Show**
- Read
- (~)

Eq

- Eq typeclass provides an interface for testing for equality.
- Any type where it makes sense to test for equality between two values of that type should be a member of the Eq class.
- If there's an Eq class constraint for a type variable in a function, the function should use **==** or **/=** somewhere inside its definition.

Ord

- Ord is for types that have an ordering
- Ord covers all the standard comparing functions such as >, <, >= and <=

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

Ord is is a subclass of Eq, i.e. any type that is an instance of Ord is also a subclass of Eq.

Enum

- `Enum` members are sequentially ordered types — they can be enumerated.

- The main advantage of the `Enum` typeclass is that we can use its types in list ranges.

- They also have defined successors and predecesors, which you can get with the `succ` and `pred` functions.

- Instances of `Enum`: `Bool`, `Char`, `Int`, `Integer`, `Double`...

```
ghci> ['a'..'e']
"abcde"

ghci> [3 .. 5]
[3,4,5]

ghci> succ 'B'
'C'
```

Bounded

Bounded members have an upper and a lower bound.

```
ghci> minBound :: Int
-2147483648

ghci> maxBound :: Bool
True
```

```
ghci> :t minBound
minBound :: Bounded a => a
```

Num

Its members have the property of being able to act like numbers.

```
ghci> :t 20
20 :: (Num t) => t

ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

Minimal definition: `(+), (*), abs, signum, fromInteger, (negate | (-))`

Show

- Members of Show can be presented as strings.
- The most important function for the Show typeclass is `show`. It takes a value whose type is a member of Show and presents it to us as a string.

```
ghci> show 3
"3"
ghci> show True
"True"
```

An Aside on Type Annotations

The show functions for `Int` and `Double` are different

We can force Haskell to read the number `3` as a `Double` by writing it `3.0`

But we can also use explicit **type annotations** wherever we want Haskell to read something as a type that might not be obvious

```
ghci> show 3
"3"
ghci> show (3 :: Double)
"3.0"
ghci> show ("three" :: Double)
error … Expected: Double
        Actual: String
```