COMP105 Lecture 13

Type Classes

Type classes

Some functions are polymorphic, but can't be applied to **any** type

```
ghci> 1 + 1
2
```

```
ghci> 1.5 + 2.5
4.0
```

```
ghci> "hello" + "there"

No instance for (Num [Char]) arising from a use of ‘+’
In the expression: "hello" + "there"
```

Type classes

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

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

`Num` is a **type class**

- It restricts the type variable a to only be number types
- `Int`, `Integer`, `Float`, `Double` are all contained in `Num`
- `Char`, `Bool`, tuples and lists are not in `Num`

Type classes

The Eq type class only allows types that can be compared with ==

```
ghci> 1 == 1
True

ghci> [1,2,3] == [4,5,6]
False

ghci> ('c', False) == ('c', False)
True

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

Type class syntax

```
equals_two a b = a + b == 2

ghci> :t equals_two
equals_two :: (Eq a, Num a) => a -> a -> Bool
```

So the syntax is

```
([Type class 1], [Type class 2], ...)  => [Type]
```

The most general type annotation

The **most general type** annotation is the one that is least restrictive

```haskell
equals_two a b  = a + b == 2


-- Works but too restrictive
equals_two :: Int -> Int -> Bool

-- Most general
equals_two :: (Eq a, Num a) => a -> a -> Bool

-- Too general (will give error)
equals_two :: a -> a -> Bool
```

Number type classes

`Num` has two **sub-classes**

Integral represents whole numbers (contains `Int` and `Integer`)

```
ghci> :t div
div :: Integral a => a -> a -> a
```

Fractional represents rationals (contains `Float`, `Double`, and `Rational`)

```
ghci> :t (/)
(/) :: Fractional a => a -> a -> a
```

Number type classes

Why does this work?

```
ghci> 10 `div` 2
5
ghci> 10/2
5.0
```

Numbers in Haskell code have a **polymorphic type**

```
ghci> :t 1
1 :: Num a => a
```

When they are used, Haskell will convert them to the correct member of `Num`

Number type classes

You can use the `::` operator to **force** a number be a particular type

```
ghci> 1 :: Integer
1

ghci> 1 :: Float
1.0

ghci> (1 :: Integer) / 2

No instance for (Fractional Integer) arising from
a use of ‘/’
```

Converting integers to numbers

Once the type has been **fixed**, it is fixed

- But you can convert back to a more generic type using `fromIntegral`

```
ghci> fromIntegral (1 :: Int) / 2
0.5
```

```
ghci> :t fromIntegral (1 :: Int)
fromIntegral (1 :: Int) :: Num b => b
```

```
ghci> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b
```

Converting floats to integers

Converting floats to integers is a **lossy** operation

```
ghci> ceiling 1.6
2

ghci> floor 1.6
1

ghci> truncate 1.6
1

ghci> round 1.6
2
```

Typeclasses that you might encounter

Haskell includes many typeclasses that we won't see on this course

```
ghci> :t length
length :: Foldable t => t a -> Int
```

`length` works on any data structure that is `Foldable`

For COMP105, if you see

- `Functor`
- `Foldable`
- `Traversable`

then think **list**

Exercises

Determine the most general type annotation for the following functions.

1.
   ```
   square_area length width = length * width
   ```

2.
   ```
   triangle_area height base = height * base / 2
   ```

3.
   ```
   equal_heads list1 list2 = head list1 == head list2
   ```