COMP105 Lecture 12

Polymorphic Types

Outline

Today

- Polymorphic types
- Type classes

Relevant book chapters

- Programming In Haskell Chapter 3
- Learn You a Haskell Chapter 3

Type polymorphism

Some functions work on **many** different types

```
length' [] = 0
length' (_:xs) = 1 + length' xs
```

```
ghci> length' [1,2,3]
3
ghci> length' "abc"
3
ghci> length' [True, False, False]
3
```

`length'` works on all lists, even though they have **different** types

Type polymorphism

So what is the type of `length'`?

```
ghci> :t length'
length' :: [a] -> Int
```

a is a **type variable**

- The function can be applied to any list
- a will represent the type of the list elements

This is called type **polymorphism**

Type variables

Type variables can appear **more than once**

```
ghci> :t head
head :: [a] -> a

ghci> :t tail
tail :: [a] -> [a]
```

These types specify that the return type will be **determined** by the type of the input

Type variables

Functions types can use **multiple variables**

```
ghci> fst (1, 2)
1
ghci> snd (1, 2)
2
```

```
ghci> :t fst
fst :: (a, b) -> a
ghci> :t snd
snd :: (a, b) -> b
```

Each variable can be bound to a **different** type

Type variables

Function types can tell you a lot about what the function **does**

```
ghci> zip [1,2,3] "abc"
[(1,'a'),(2,'b'),(3,'c')]
```

```
ghci> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

Type annotations

It is good practice to give **type annotations** for your functions

```
length' :: [a] -> Integer
length' [] = 0
length' (_:xs) = 1 + length' xs
```

The syntax is

```
[function name] :: [type]
```

The annotation is usually placed **before** the function definition

Type annotations

If you don't give a type annotation, then Haskell will **infer** one for you

```
all_true [] = True
all_true (x:xs) = x && all_true xs
```

```
ghci> :t all_true
all_true :: [Bool] -> Bool
```

- The input must be a list (due to the use of `:`)
- The list must contain `Bool`s (due to the use of `&&`)

Type annotations

Annotating your functions can make it easier to **catch bugs**

```
third_head list = head (head (head list))

ghci> :t third_head
third_head :: [[[a]]] -> a
```

```
third_head :: [a] -> a
third_head list = head (head (head list))
```

```
Couldn't match type ‘a’ with ‘[[a]]’
  ‘a’ is a rigid type variable bound by
    the type signature for third_head :: [a] -> a
```

Exercise

Which types are returned by the following queries?

```
ghci> :t take
```

```
ghci> :t (:)
```

```
ghci> :t (++)
```

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

```
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`

Summary

- Type polymorphism
- Type classes

Next time: Higher order functions