COMP105 Lecture 13

Polymorphic Types

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 [] = False
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

What types are returned by the following queries?

```
ghci> :t take
```

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

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