COMP105 Lecture 12

Function Types

Function types

Functions also have a type

```
is_lower c = c `elem` ['a'..'z']
```

```
ghci> is_lower 'a'
True

ghci> :t is_lower
is_lower :: Char -> Bool
```

Function types

For a **one** argument function, the type is written as

```
[input type] -> [output type]
```

Examples:

- `Bool -> Bool`
- `[Char] -> Char`
- `(Int, Int) -> Int`

Function types

For a function with **more than one** argument, the type uses multiple ->

```
boolean_and a b = a && b
```

```
ghci> :t boolean_and
boolean_and :: Bool -> Bool -> Bool

ghci> :t (||)
(||) :: Bool -> Bool -> Bool
```

Function types

So the function type with **multiple arguments** are written as

```
[arg1 type] -> [arg2 type] -> ...  -> [return type]
```

Examples

- `Int -> Int -> Int`
- `[Char] -> Int -> Char`
- `Bool -> Bool -> Bool -> Bool`

Partial application

In most functional languages, functions can be **partially applied**

```
plus a b = a + b

plus2 = plus 2
```

```
ghci> plus2 2
4
ghci> plus2 10
12
```

This is also known as **currying**, named after Haskell Curry

Partial application

In partial application

- We fix **some** of the arguments
- We leave other arguments unfixed

The creates a **new function** that only has the unfixed arguments

```
func a b c = "Arguments are: " ++ [a, b, c]

func2 = func 'x'
```

```
ghci> func2 'y' 'z'
"Arguments are: xyz"
```

Partial application

```
func a b c = "Arguments are: " ++ [a, b, c]

func2 = func 'x'
```

```
ghci> :t func
func :: Char -> Char -> Char -> [Char]

ghci> :t func2
func2 :: Char -> Char -> [Char]
```

Partial application

Partial application must follow the **argument order**

```
join_three x y z = x ++ [' '] ++ y ++ [' '] ++ z
```

```
f1 = join_three
f2 = join_three "hello"
f3 = join_three "hello" "to"
f4 = join_three "hello" "to" "you"
```

Some more examples

```
pow2 = (^) 2

ghci> pow2 10
1024
```

```
first_four = take 4

ghci> first_four [1..10]
[1,2,3,4]
```

```
prepend1 = (:) 1

ghci> prepend1 [1,2,3]
[1,1,2,3]
```

Partial application of infix operators

Infix operators can be partially applied **on both sides**

```
ghci> let f = (/2)
ghci> f 4
2.0
```

```
ghci> let g = (1/)
ghci> g 4
0.25
```

Bracketing for function types

Function application should be thought of **multiple** partial applications

```
multThree x y z = x * y * z
```

```
ghci> ((multThree 2) 3) 4
24
```

This means that the function type brackets to the **right**

```
Int -> Int -> Int -> Int
```

is the same as

```
Int -> (Int -> (Int -> Int))
```

Multiple arguments vs Tuples

Previously we've seen that you can write functions in two ways

- Using the usual "spaces" syntax
- Using a tuple

```
multThree x y z = x * y * z

multThree' (x, y, z) = x * y * z
```

These both do the same thing, but the second version **cannot** be partially applied

- It's best to avoid tuples unless they are necessary

Exercises

Without using `:type`, what are the types of the following functions?

1. `isA c = c == 'a'`

2. `isADouble c1 c2 = (isA c1, isA c2)`

3. `exclaim = (++"!")`