Session 3: Functions

COMP2221: Functional programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

Recap

- Defined and motivated types
- Different concepts of typing (dynamic/static)
- Static vs. dynamic type checking
- Type checking mechanisms
- Considered basic data types in Haskell
  - `Bool`
  - `Int`, `Integer`, `Double`
  - `Char`
- Defined *lists* `[a]` and *tuples* `(a, b, c)`
- Used tuples and lists to model functions with multiple input parameters

Functions have types

- Functions have types in all programming languages, Haskell makes this particularly explicit

Functions of one argument "unary"

Map from one type to another

```haskell
not :: Bool -> Bool
```

Functions of two arguments "binary"

Map from two types to another

```haskell
add :: (Int, Int) -> Int
```

An alternative way to define binary functions

- Since functions are *first class objects*, functions of *more than one* argument are typically written in Haskell as *functionals*
- Functionals are functions that return other functions
- Naturally extends from binary to n-ary functions

Currying

"Curried" view of binary functions

```haskell
add :: Int -> (Int -> Int)
```

"add takes an Int and returns a function which takes an Int and returns an Int"

Definition (Currying (informal))

Turning a function of n arguments into a function of $n - 1$ arguments.

Excursus: history of currying

- Idea first introduced by Gottlob Frege
- Developed by Moses Schönfinkel in the context of combinatory logic
- Further extended by Haskell Brooks Curry working in logic and category theory
- Name “currying” coined by Christopher Strachey (1967)

Demo time

Let's look at currying in Haskell

Currying conventions

- (Almost) all functions in Haskell are written in *curried* form

⇒ To avoid messy syntax, this leads to associativity rules for `->` and function application.

`->` associates to the right

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

Function application associates to the left

```
mult x y z
-- Means
((mult x) y) z
```

Purpose of currying

- *Easier* to reason about and prove things with functions of only one variable
- Flexibility in programming: makes composing functions simpler
- Related to *partial evaluation* where we bind some variables in an n-ary function to a value

$\Rightarrow$ Currying allows for functions with multiple arguments in languages that only support unary functions such as Haskell and the *Lambda Calculus*

Lambda expressions in Haskell

Nameless functions

- As well as giving functions names, we can also construct them *without* names using *lambda expressions*

```
-- The nameless function that takes
-- a number x and returns x + x
\x -> x + x
```

- Use of λ for nameless functions comes from *lambda calculus*, which is a theory of functions.
- There is a whole formal system on reasoning about computation using λ calculus (developed by Alonzo Church in the 1930s) $\Rightarrow$ a different course
- It is also a way of formalising the idea of *lazy evaluation* (on which more later)

Use cases for unnamed functions I

- Formalises idea of functions defined using currying

```haskell
add x y = x + y
-- Equivalently
add = \x -> (\y -> x + y)
```

- The latter form emphasises the idea that `add` is a function of one variable that returns a function
- Also useful when returning a function as a result

```haskell
const :: a -> b -> a
const x _ = x
-- Or, perhaps more naturally
const x = \_ -> x
```

"`const` eats an `a` and returns a function which eats a `b` and always returns the same `a`."

Use cases for unnamed functions II

- What good is a function which always returns the same value?
- Often when using *higher-order* functions, we need a base case that always returns the same value.

```
length' :: [a] -> Int
length' xs = sum (map (const 1) xs)
```

  "The length of a list can be obtained by summing the result of calling `const 1` on every item in the list"

- We will see some more of this when we look at *higher order* functions.

Use cases for unnamed functions III

- Also useful where the function is only used once

```haskell
-- Generate the first n positive odd numbers
odds :: Int -> [Int]
odds n = map f [0..n-1]
  where
    f x = x*2 + 1
```

- Can be simplified (removing the `where` clause)

```haskell
odds :: Int -> [Int]
odds n = map (\x -> x*2 + 1) [0..n-1]
```

Translating between the two forms

- It is always possible to translate between named functions and arguments, and the approach using λ expressions of one argument
- Just move the arguments to the right hand side and put it inside a λ, repeat with remainder until you're done.

```
f a b c = ...
-- Move formal arguments to right hand side with a lambda
f = \a b c -> ...
-- move remaining arguments into new lambdas
f = \a -> (\b -> (\c -> ...))
```

- Which option fits more naturally is often a style choice

Function syntax conventions

Syntax conventions

- Function application is *so important* that it is written as quietly as possible: with whitespace
- *All* functions can be called in *prefix* form:
  "`foo a b`", not "`a foo b`"
- . . . but, special syntax for binary functions.

Binary functions: infix notation

Infix notation

All binary functions (which have type `a -> b -> c`) can be written as *infix* functions.

Symbol only names

Names consisting *only* of symbols (e.g. `+`, `*`)

```
1 + 2   -- infix notation
(+) 1 2 -- prefix notation
False && True   -- infix notation
(&&) False True -- prefix notation
```

"Normal" names

Names with alpha-numeric characters (e.g. `div`, `mod`)

```
mod 3 2   -- prefix notation
3 `mod` 2 -- infix notation using backticks
```

Summary

- Learned that functions have types
- Discussed currying as a manner to define functions with multiple arguments
- Introduced the lambda calculus and the idea of anonymous functions
- Saw syntax for these λ expressions in Haskell
- And how they can formalise (or make it easier to read) curried functions:

```haskell
add x y = x + y
-- vs
add = \x -> (\y -> x + y)
```

- Considered infix and prefix notation