FMFP - Lecture Notes Week 4

Ruben Schenk, ruben.schenk@inf.ethz.ch

March 16, 2022

0.0.1 Partial Application

Functions of multiple arguments can be **partially applied.** Consider the following example:

```
multiply :: Int -> Int -> Int
multiply a b = a * b

? :type multiply 7
Int -> Int

? :type map
(a -> b) -> [a] -> [b]

? map (multiply 7) [1, 2, 3, 4]
[7, 14, 21, 28] :: [Int]
```

It is important to note here that each function takes *exactly one argument!* Consider `multiply :: Int -> Int -> Int` means `multiply :: Int -> (Int -> Int)`. Therefore, the application `multiply 2 3` means `(multiply 2) 3`.

Furthermore, we might use **tuple arguments.** They may are equivalent to multiple-argument functions, however they do no not allow partial application!

1 Higher-Order Programming and Types

1.1 Overview

1.1.1 Implement a Function with foldr

1. Identify the recursive argument and static and dynamic arguments

```
mystery a b c []       = a + b - c
mystery a b c (x : xs) = mystery x (b + c) c xs
```

2. Write a helper with only recursive (first) and dynamic arguments

```
aux []       a b = a + b - c
aux (x : xs) a b = aux xs x (b + c)
```

3. Move the dynamic arguments to the right of the equals

```
aux []       = \a b -> a + b - c
aux (x : xs) = \a b -> aux xs x (b + c)
```

4. Rewrite aux using `foldr` replacing `aux xs` with local variable `rec`

```
aux = foldr (\x rec a b -> rec x (b + c)) (\a b -> a + b - c)
```

5. Inline `aux`

```
mystery a b c xs =
  foldr (\x rec a b -> rec x (b + c)) (\a b -> a + b - c) xs a b
```