FMFP - Lecture Notes Week 4

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

April 1, 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
   ```

1.2 Case Study: Operations on Vectors and Matrices

Vectors and vector addition can be easily defined by:

```
type Vector = [Int]

vecAdd :: Vector -> Vector -> Vector
```

```
vecAdd (x:xs) (y:ys) = (x + y) : vecAdd xs ys
vecAdd _             = []
```

We could also use `zipWith`, which is a combination of `map` and `zip`. This would look as follows:

```
vecAdd :: Vector -> Vector -> Vector
vecAdd = zipWith (+)
```

An $n \times m$ **matrix** can be represented *column-wise* using lists. We might write this like:

```
type Matrix = [Vector]

matAdd :: Matrix -> Matrix -> Matrix
matAdd = zipWith vecAdd
```

Some other matrix-related definitions:

```
-- Constant vector of size n
vconst :: Int -> Int -> Vector
vconst 0 _ = []
vconst n x = x : vconst (n - 1) x

-- unit matrix of size n x n
unit :: Int -> Matrix
unit 0 = []
unit n =
    (1 : vconst (n - 1) 0)
    : map (0:) (unit (n - 1))
```

Transposing of a matrix can be implemented as follows:

```
tr :: Matrix -> Matrix
tr []      = []
tr [v]     = map (\x -> [x]) v
tr (v:vs)  = zipWith (:) v (tr vs)
```

Another very important operation in linear algebra is the **dot product.** We propose different ways to implement it in Haskell:

```
-- Version 1: Loop / accumulator
skProd :: Vector -> Vector -> Int
skProd xs ys = loop xs ys 0
    where
        loop []     []     0 = p
        loop (x:xs) (y:ys) p = loop xs ys (x * y + p)

-- Version 2: Explicit recursion
skProd :: Vector -> Vector -> Int
skProd (x:xs) (y:ys) = x * y + skProd xy ys
skProd _      _      = 0

-- Version 3: Using library functions
skProd :: Vector -> Vector -> Int
skProd v w = sum (zipWith (*) v w)
```

Finally, we can go to the most interesting problem: **matrix multiplication.** WE first start by multiplying an $n \times m$ matrix A with vector b of size m, which is equivalent to the scalar product of A's rows (i.e. the columns of `tr` A) with b:

```
vecMult :: Matrix -> Vector -> Vector
vecMult a b = map ('skProd' b) (tr a)
```

With this problem solved, matrix multiplication simply iterates `vecMult` A over an $m \times k$ matrix B:

```
matMult :: Matrix -> Matrix -> matrix
matMult a b = map (vecMult a) b
```