Programming Paradigms 2024
Session 6 : Higher-order functions

Hans Hüttel

22 October 2024

Our plan for today

1. The learning goals
2. Presentations of the discussion problems
3. Problem no. 1
4. Problem no. 2
5. Break
6. Problem no. 3
7. Problem no. 4
8. Problem no. 5
9. If time allows: More problems at your own pace.
10. We evaluate today's session – please stay until the end!

Learning goals

- To be able to explain precisely what a higher-order function is and what the type of a higher-order function is
- To be able to explain precisely higher-order functions on lists including map, filter , foldr and foldl
- To be able to explain and understand the recursive definitions of common higher-order functions on lists
- To be able to use higher-order functions, including map, filter , foldr and foldl , for solving programming problems in Haskell
- To be able to explain and use function composition in Haskell

Discussion problem – positions

Every letter in the lowercase English alphabet has a position. "a" has position 1, "c" has position 3 and "h" has position 8.

In Haskell, every string is a list of characters. So String is the same type as [Char].

We can define a function positions that, given a string of lowercase letters str gives us the list of positions of the characters in str.

As an example, positions "abba" gives us [1,2,2,1] . Use the higher-order functions in Chapter 7 to define positions.

Discussion problem – sumsq

The function sumsq takes an integer n as its argument and returns the sum of the squares of the first n integers. So sumsq n returns the sum

$$1 + \ldots + n^2$$

As an example, sumsq 4 gives us 30 and sumsq 9 gives us 285 . Use foldr to define sumsq – and do not use map.

About the problems

- There is a problem set with the problems that we talk about in class. You can download it from the course page on Moodle.
- *If you finish early*, work on one of the "lettered problems" in the problem set instead of working on the next numbered problem.
- You have to have access to a computer with an installation in order for class activities to be meaningful for you. Looking at your phone or watching someone else will not help you learn. Now is the time for you to program on your own.

Discussion – dbs

A presidential candidate calls himself "the greatest programmer alive". To impress a journalist, he defined a function dbs over lists using higher-order functions. The list should take a list of pairs of numbers and return the list of tuples in which the second element is twice the first element. As an example dbs [(1,4) ,(3,6) ,(9,2) ,(4,8)] should return [(3,6) ,(4,8)].

Here is what the presidential candidate wrote.

```
dbs xs | length xs == 0 = []
       | otherwise = filter ((2 * (fst(head xs
         ))) == (snd(head xs))) xs
```

The journalist remarked that something was not quite right and that the coding style was rather clumsy. Comments, please!
The definition can be found on Moodle in the file `dbs.hs`.

Problem 1 - dbs (10 minutes)

Now it is your turn! Write a sensible, non-clumsy definition of dbs.

Problem 2 – within (15 minutes)

The within function takes a list of numbers and a pair of numbers, returns a list of numbers which are in the input list and within the range (inclusive) given by the input pair.

The elements in the output list appear be in the same order they appeared in the input list. If the input pair is (n1,n2), assume that n1 is the lower bound of the range and n2 is the upper bound of the range.

As an example, within [1,3,4,5,2] (1,3) should give us [1,3,2] and within [1,3,4,5,2] (3,1) should give us [] .

Define within using the higher-order functions in Chapter 7.

Problem 3 – sumrows (10 minutes)

Implement the sumrows function. The function takes a list of number lists returns a one-dimensional list of numbers with each number equal to the sum of the corresponding row in the input list. If a list is empty, its sum is 0.

As an example, sumrows [[1,2], [3,4]] should give us [3, 7], and sumrows [[],[],[1]] should give us [0,0,1] .

Define sumrows using the higher-order functions in Chapter 7.

Problem 4 – It's only natural (15 minutes)

The base of the natural exponential function $e = 2.718\ldots$ can be written as the limit of the infinite series

$$\sum_{k=0}^{\infty} \frac{1}{k!}$$

The function approx should give us the approximation of e that we find by adding the first n terms of this infinite series. That is,

$$\text{approx } n = \sum_{k=0}^{n} \frac{1}{k!}$$

Define approx using the higher-order functions in Chapter 7; the factorial function $k!$ is defined by

```
fact k = product [1..k]
```

Problem 5 – fingo (10 minutes)

Here is a function. What does it do? And why? Use the explanation at the very end of Section 7.3 in the book to help you answer the "why"-question.

```
fingo :: [a] -> [a] -> [a]

fingo xs ys = foldr (:) xs ys
```

Evaluation

- What did you find difficult?
- What surprised you?
- What went well?
- What could be improved? How does the setup work this time?
- Is there a particular problem that we should follow up on with a short video?