Programming Paradigms
First session about typed functional programming in Haskell
Suggested solutions

Hans Hüttel

20 October 2020

Problem 1

The goal of this problem is to write a Haskell function fib that finds the nth Fibonacci number.

1. Specify the type of fib without using the Haskell system.

   Solution:

   ```
   fib 0 = 0
   fib 1 = 1
   fib n = fib (n-1) + fib (n-2)
   ```

   Solution: We have

   ```
   fib :: (Num t) => t -> t
   ```

2. Write the function. Use it to find fib 3 and try to find fib 45. What is the problem here?

   Solution: The problem is that the call fib 45 causes the Haskell runtime system to hang. As we see below, this is due to the time complexity of the fib function.

3. The time complexity of fib should be measured as the number of additions as a function of n needed to computed fib(n) for any given n. What is the time complexity of fib? Justify your answer.

   Solution: Let $T(n)$ denote the time complexity of computing fib(n) measured as the number of function calls needed. We get the recurrence equations

   $$\begin{aligned} T(0) &= 1 \\ T(1) &= 1 \\ T(n) &= T(n-1) + T(n-2) \end{aligned}$$

   which is exactly the same as the definition of fib. As we know that fib(n) $= O(2^n)$ (this can be seen from the closed form expression for the Fibonacci function) we know that this is also the case for $T(n)$; in other words, the time complexity is exponential.

Problem 2

The goal of this problem is to write a Haskell function **reverse** that will reverse a list such that e.g. **reverse** [1,2,3] evaluates to [3,2,1].

1. Write the function without first specifying its type.

   Solution:

   ```
   rev [] = []
   rev (x:xs) = (rev xs) ++ [x]
   ```

2. Find the type of **reverse** without using the Haskell system. Justify your answer. Use the Haskell system to check if your answer is correct.

   Solution: The type is

```
rev :: [t] -> [t]
```

Problem 3

A *palindrome* is a string that is the same written forwards and backwards such as “Otto” or “Madam”.

The goal of this problem is to write a Haskell function ispalindrome that will determine if a string of characters is a palindrome.

1. First specify the type of ispalindrome without using the Haskell system.

   Solution: The type is

   ```
   ispalindrome :: (Eq t) => [t] -> Bool
   ```

2. Now write the function.

   Solution: A list is a palindrome, if it is equal to its own reverse.

   ```
   ispalindrome l = (l == rev l)
   ```

Problem 4

A theorem in number theory states that every non-zero real number x can be written as a *continued fraction*. This is a potentially infinite expression of the form

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cfrac{1}{\cdots \frac{1}{a_n}}}+}}} \tag{1}$$

For rational numbers, the a_i's will eventually all be 0, so the continued fraction is finite; for irrational numbers, the continued fraction will be infinite. See e.g. [1] for more.

The goal of this problem is to write a Haskell function cfrac that will, given a real number r and a natural number n, finds the list of the first n numbers in the continued fraction expansion of r.

1. First specify the type of cfrac without using the Haskell system.

   Solution: The type should be

   ```
   cfrac :: (Eq a, Integral t, Num a, RealFrac a1) => a1 -> a -> [t]
   ```

2. Now write the function.

   Solution:

   ```
   cfrac r 0 = []
   cfrac r n = a : (remainfrac r1)
               where
                   a = truncate r
                   r1 = (r - (fromIntegral a))
                   remainfrac r1 = cfrac (1/r1) (n-1)
   ```

Problem 5

The goal of this problem is to write a Haskell function **last** that finds the last element of a list.

1. First specify the type of **last** without using the Haskell system.

   Solution: The type should be

   ```
   last :: [t] -> [t]
   ```

2. Now write the function.

```
last' (x:[]) = x
last' (x:xs) = last' xs
```

Problem 6

The goal of this problem is to write a Haskell function flatten that will flatten a twice-nested list. For instance, we should get that flatten [[1,2,3], [3,2],[],[7,8,2]] evaluates to [1,2,3,3,2,7,8,2]

1. First write the function.

   Solution:

   ```
   flatten [] = []
   flatten (x:xs) = x ++ (flatten xs)
   ```

2. Now find the type of flatten without using the Haskell system. Justify your answer. Use the Haskell system to check if your answer is correct.

   Solution: The type is

   ```
   flatten :: [[t]] -> [t]
   ```

A problem directly related to the miniproject

An *association list* is a representation of the graph of a finite function f as a list of pairs $[x_1, f(x_1), \ldots, x_n, f(x_n)]$. For instance, the function defined by

$$\begin{aligned} f(1) &= \mathsf{false} \\ f(2) &= \mathsf{true} \\ f(3) &= \mathsf{false} \\ f(4) &= \mathsf{true} \end{aligned}$$

can be represented by the association list [(1,false),(2,true),(3,false),(4,true)]. We say that an association list is valid if it describes the graph of a function, that is, every argument is bound to precisely one value in the list. So for a list to be valid, whenever (x_i, y_i) and (x_j, y_j) are both found in the list, then $x_i \neq x_j$.

The goal of this problem is to write three Haskell functions valid, findfun and **lookup** with the following behaviour

- valid will tell us if a list of pairs is a valid association list.
- findfun will return the function associated with an association list.
- **lookup** will, given an association list l and an argument x find the function value of x if it exists

For each of these functions, you should specify its type before writing any code.

Please note: In the next session, we will use the **Maybe** type constructor to deal with returning a non-value if a meaningful value does not exist. For now, it is perfectly fine to ignore this issue.

Bibliography

[1] Wikipedia. Continued fractions. `https://en.wikipedia.org/wiki/Continued_fraction`.