Programming Paradigms
Second session about typed functional programming in Haskell

Hans Hüttel

27 October 2020

Read this first: How to write your answers

At the written exam, you are supposed to write your answers by annotating a PDF file. So you should also do this now, during the course itself. to practice how this is done. Use the annotation capabilities of your PDF reader to do this. Several commonly available PDF reader support this. If in doubt, ask me.

Write your answers in the boxes provided *except for the last problem that relates directly to your miniproject*. Here you should save your answer in a separate file that you can use when completing the miniproject.

Please also read this

One of the very important learning goals of this session is that you become able to find the types of expressions in Haskell. In some of these problems, you are asked first specify the type of the functions and other expressions that are being asked for without consulting the Haskell interpreter. Only then should you start programming.

In other problems, you should do the opposite – first write the function and try to find its type. In these cases, do not just make a guess – please *provide the reasoning behind your answer*.

Problem 1

Use list comprehension to define a function isprime that will tell us if any given natural number is a prime number.

- Figure out what the type of isprime should be.

```
isprime :: Integral a => a -> Bool
```

- Now write the code. *Hint:* By definition, a natural number n is a prime number if its only divisors are 1 and n. It is a good idea to define an auxiliary function that returns the list of divisors of n using list comprehension.

```
isprime n = ( [ x  | x <- [1..n], n `mod` x == 0 ] == [1,n] )
```

Problem 2

The Peano axioms define the set of natural numbers as follows.

- Zero is a natural number
- The successor of a natural number is a natural number

1. Use this to define a recursive datatype Nat for the natural numbers.

```
data Nat = Zero | Succ Nat
```

2. Define a function nat2int that will convert any member of Nat to its corresponding integer. First figure out what the type of nat2int should be, and only then write the code.

```
nat2int Zero = 0
nat2int (Succ n) = (nat2int n) + 1
```

3. Define a function int2nat that will convert any integer to its corresponding member of Nat. First figure out the type of int2nat in the Haskell type system, and only then write the code.

```
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Problem 3

In the previous session, we looked at the Fibonacci numbers and an inefficient way of computing fib n for any n.

Now define an efficient version using lazy evaluation that will allow you to compute fib 40.

Hint: Define a function that returns the list of all Fibonacci numbers, computed from below, starting with 1 and 1.

```
fib n = head (drop (n-1) (fibs 1 1))
        where
                fibs a b = a : (fibs b (a+b))
```

Problem 4

Consider the following data type of binary trees:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)}
```

We say that a binary tree is *balanced* if the number of leaves in every left and right subtree differs by at most one with leaves themselves being trivially balanced. Define a function balanced that will tell us if a binary tree is balanced or not.

1. Write the definition of the function balanced. *Hint:* It is probably a good idea to define an auxiliary function that computes the number of leaves in a binary tree.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

number_of_leaves (Leaf n) = 1
number_of_leaves (Node t1 t2) = let
                                    v1 = number_of_leaves t1
                                    v2 = number_of_leaves t2
                                in
                                    v1+v2

balanced (Leaf n) = True
balanced (Node t1 t2) = let
                                    v1 = number_of_leaves t1
                                    v2 = number_of_leaves t2
                                    diff = (abs v1 - v2)
                        in
                                (diff <= 1) && balanced t1 && balanced t2
```

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

```
balanced :: Tree a -> Bool
```

Problem 5

Without looking at the definition from the standard prelude, define the function **any** that decides if any elements of a list satisfy a predicate.

For instance, if

odd x = ((x ‘**mod**‘ 2) == 1)

then

any odd [2,5,8,3,7,4]

should return **True**, whereas

any odd [2,8,42]

should return **False**.

1. What should the type of **any** be?

I call the function any' to avoid confusion with the function of the same name in the prelude. We have

```
any' :: (t -> Bool) -> [t] -> Bool
```

2. Write a recursive definition of **any**.

```
any' f [] = False
any' f (x:xs) = (f x) || (any' f xs)
```

3. Write a definition that uses **foldl** or **foldr**.

Here are two versions:

```
any'' f = foldr (\x a -> (f x) || a) False

any''' f = foldl (\a x -> (f x) || a) False

any'''' f = foldr (\x a -> if (f x) then True else a) False
```

Problem 6

Define **filter** using **foldr**. *Hint:* How did we define **map** using **foldr**?

```
filter' pred = foldr (\x acc -> if (pred x) then x:acc else acc) []
```

A problem directly related to the miniproject

1. Create a type LTree of binary trees whose leaves are labelled with characters and whose root is labelled with a real number. *Hint:* Use a datatype for the tree part and a type synonym to define root-labelled trees.

2. Define a function treemerge that takes two trees of type LTree and returns a new tree of type LTree whose root is labelled with the sum of the roots and has two trees as subtrees.