Programming Paradigms
Written exam

Aalborg University

10 January 2024
10:00 - 13:00

You must read the following before you begin!!!

What is this? This problem set consists of 6 problems. The problem set is part of a zip archive together with a file called `solutions.hs`.

How do I begin? *Please do the following immediately!!* Unzip the zip archive – it is not enough to look within it; you *must* unzip it and save the problem set and `solutions.hs` locally on your computer. Otherwise, the answers that you write in `solutions.hs` will not be saved.

Please add your full name, AAU mail address and study number to the top of your local copy of `solutions.hs` saved to your computer where indicated in the file. If you experience problems with the file extension `.hs`, then rename the file while working and give it the file extension `.hs`, just before you submit it.

Må jeg skrive på dansk? Ja, det må du gerne. You can write your answers in Danish or in English.

How and where should I write my solution? Please write your answers by adding them to the local copy of `solutions.hs` on your computer. You must indicate in comments which problem your text concerns and which subproblem it concerns. All other text that is not runnable Haskell code (such as code that contains syntax errors or type errors) must also be written as comments.

Use the format as exemplified in the snippet shown below.

```
-- Problem 2.2

bingo = 17

-- The solution is to declare a variable called bingo with value 17.
```

How should I submit my solution? *Submit the local copy of `solutions.hs` that your solutions appear in and nothing else.* DO NOT SUBMIT A ZIP ARCHIVE.

What can I consult during the exam? During the exam, you are only allowed to use the textbook *Programming in Haskell* by Graham Hutton, your own notes and your installation of the Haskell programming environment. *GitHub CoPilot, ChatGPT and other AI-based tools are not allowed.*

What can I use for my code? You are only allowed to use the Haskell `Prelude` for your Haskell code, unless the text of a specific problem specifically mentions that you should also use another specific module. Do not use any special GHCi directives.

Is there anything else I must know? Yes. Please read the text of each problem ***very carefully*** before trying to solve it. All the information you will need is in the problem text. Please make sure that you understand what is being asked of you; it is a very good idea to read the text more than once.

Problem 1 – 16 points

1. Define a function rotate which places the head of a list at the end of the tail of the list. We expect that rotate [1, 2, 3] = [2, 3, 1] and that rotate "eat" = "ate". Is the function polymorphic?

   If yes, is the polymorphism parametric, ad hoc or both? You must justify your answer.

2. Use recursion and the rotate function to define a function allrotates that produces all the rotations of a list. We expect that allrotates [1, 2, 3] = [[1, 2, 3], [2, 3, 1], [3, 1, 2]].

   Is the function polymorphic? If yes, is the polymorphism parametric, ad hoc or both? You must justify your answer.

3. Give another definition of allrotates called allrotates' that is not recursive but uses either map or foldr as well as rotate.

Problem 2 – 18 points

A *partially labelled* a*-tree* is a binary tree in which each internal node can be either labelled with an element of type a or unlabelled, but leaves are always labelled with elements of the type a. A partially labelled a-tree is said to be *fully labelled* if every node is labelled.

Figure 1 shows two partially labelled trees. t_1 is not fully labelled, but t_2 is fully labelled.

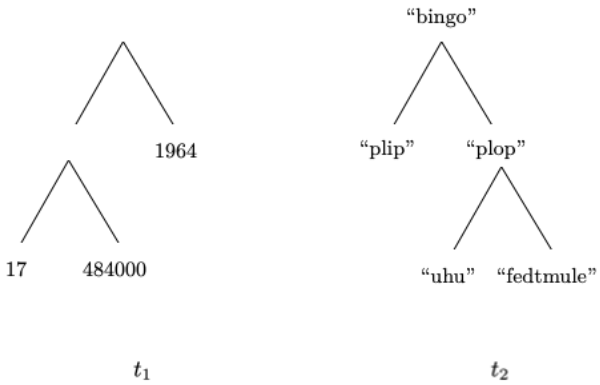

Figure 1: A partially (but not fully) labelled Integer-tree and a partially (and fully) labelled String-tree.

1. Define a datatype Tree a that describes partially labelled a-trees. Show how one represents the trees t_1 and t_2 in Figure 1 using your datatype.

2. Define a function isfull with type isfull :: Tree a −> Bool such that isfull t evaluates to True if t is fully labelled and isfull t evaluates to False otherwise. isfull t_1 should return False, and isfull t_2 should return True.

3. Define a function preorder of type preorder :: Tree a −> Maybe [a] which lists the nodes of a fully labelled tree in preorder but returns Nothing if the tree is not fully labelled. preorder t_1 should return Nothing, and preorder t_2 should return Just ["bingo","plip","plop","uhu","fedtmule"].

   You definition must satisfy the following requirements:

   - The definition *does not* use the isfull function that you have just defined or any other helper function.
   - The definition uses the Maybe monad and do-notation.

Problem 3 – 16 points

Define a function remove which takes two strings as its arguments and removes every letter from the second list that occurs in the first list.

As an example, remove "first" "second" should give us the string "econd".

1. Define remove using list comprehension.

2. Define remove using recursion *without using* list comprehension.

Problem 4 – 18 points

Here is the declaration of a type WrapString and a declaration that makes it an instance of Functor.

```
newtype WrapString a = WS (a,String) deriving Show

instance Functor WrapString where
   fmap f (WS (x,s)) = WS (f x,s)
```

1. Extend the above piece of code with an instance declaration such that WrapString becomes an applicative functor also.
2. Extend the above piece of code with an instance declaration such that WrapString becomes a monad also.
3. Use a do-block in the WS-monad that you now have to define a function pairup such that we have that pairup (WS (4,"horse")) (WS (5,"plonk")) gives us WS ((4,5),"horse").

Problem 5 – 16 points

Here are four types. For each of the four cases, find an expression or function definition in Haskell that has this particular type and explain if this case involves polymorphism and, if it does, whether it is parametric polymorphism, ad hoc polymorphism or both.

1. (Ord a, Num a) => a -> a -> a -> (a, a)
2. [(Integer, p -> Char)]
3. (t1 -> Bool -> t2) -> t1 -> t2
4. (Num a, Enum a) => [a]

Problem 6 – 16 points

1. Give a *recursive* definition of the list naturals of natural numbers. One is the least natural number.
2. Use map to define an infinite list facs such that the $i + 1$th element in facs is $i!$, the factorial of i. Thus, we expect that take 10 facs is

   [1,1,2,6,24,120,720,5040,40320,362880]

3. Give a recursive definition of the infinite list of factorials, called facs', that does not use a definition of the factorial function but uses the zipWith function from the Haskell prelude. The zipWith function is defined as

```
zipWith f xs [ ] = [ ]
zipWith f [ ] ys = [ ]
zipWith f (x:xs) (y:ys) = (f x y) : zipWith f xs ys
```