Programming Paradigms
Written exam

Aalborg University

6 January 2025
9:00 - 12:00

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.

Is Internet access allowed during the exam? You are only allowed to use Internet access for getting this problem set and for submitting your answers to Digital Eksamen.

What am I allowed to consult during the exam? During the exam, you are only allowed to use the textbook *Programming in Haskell* by Graham Hutton, notes that you have written yourself and your local 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*.

Are the external examiners present during the exam? No. The external examiners are never present during a written exam.

Problem 1 – 16 points

Below are four types. For each type, find an expression or definition that will have this particular type as its most general type *if you use type inference*. In each case, explain if the expression or definition is polymorphic and if it is, in which way it is polymorphic.

1. a -> b -> [(a, b)] -> [(a, b)]
2. Num a => a -> a -> (a, [a])
3. (Ord a, Num a) => ([Char], a -> Bool)
4. [[Char] -> [Char]]

Problem 2 – 18 points

The goal of this problem is to represent the structure of software components. A software component can have zero or more subcomponents that are also software components (that can in turn have subcomponents etc.). Components with no subcomponents are called *primitive*. Every component has a name which is a string and a *root size*. The root size must be a whole number.

Figure 1 presents two diagrams that show software components.

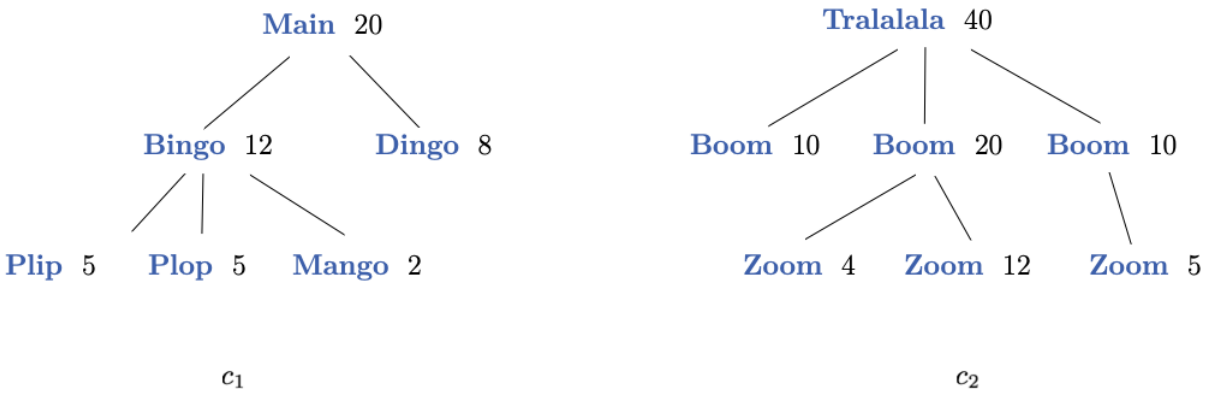

Figure 1: Software components c_1 (left) and c_2 (right).

1. Define a datatype SWComponent of software components that describes this.
2. Represent the components in the figure as values of type SWComponent.
3. A software component is valid if it is *either* primitive with a root size that is a positive whole number *or* is not primitive and with a root size that is a positive whole number equal to the sum of the root sizes of its immediate subcomponents and each of these subcomponents is also valid. We have that c_1 is valid and that c_2 is not valid.

   Define a function valid that will tell us if a software component is valid.

Problem 3 – 16 points

The National Union of Sleepwalkers have decided to cast a vote among their members in order to decide if the annual membership fee should be increased. Each member must vote and can either vote Yes, No or Abstain.

1. Define a datatype Vote that describes that votes can be of this form.

2. The result of the referendum is a list of votes. A vote is *successful* if more members voted Yes than No. Abstain votes do not count.

   Use *recursion* to define a function success that, given the result of the vote, will tell us if the vote was successful.

3. Use *list comprehension* to give another definition of success called success'.

4. Use foldr to give yet another definition of success called success''.

Problem 4 – 20 points

Here is a datatype definition. The intention is that we can use it to tell if the application of a function on real numbers results in a meaningful result and that, if that is not the case, the value of the argument that led to an error.

```
data Err a = Result a | Wrong Float deriving Show
```

The goal is now to turn Err into a monad such the we can compute logarithms and sums of logarithms in a safe way.

The logarithm function (with base e) is called log in the Haskell prelude. From mathematics we know that the logarithm $\log x$ is only defined if x is a positive real number.

1. Define a function safelog :: Float -> Err Float such that safelog x gives us Result r where $r = \log x$ if $x > 0$ and Wrong x if $x \leq 0$.

2. Define Err a to be a functor.

3. Define Err a to be an applicative functor.

4. We now define

   ```
   instance Monad Err where
       return = pure
       (>>=) (Result x) g = g x
       (>>=) (Wrong x) g = Wrong x
   ```

   The goal is now to use safelog and do-notation to define a function safesum :: Float -> Float -> Err Float.

   The intention is that safesum will compute the sum of the logarithms of two numbers, if this result is defined. Otherwise, we get the value of the first of the two arguments for which the logarithm is not defined.

   Thus, safesum 5 4 should give us Result 2.9957323. We should have that safesum (−5) 0 gives us Wrong (−5.0) and that safesum 5 (−4) gives us Wrong (−4.0).

   The definition of safesum must make use of the fact that Err a is a monad. Answers that do not make use of this will get no credit.

Problem 5 – 14 points

1. Define, using recursion, an expression odds whose value is the infinite list of odd natural numbers.
2. Use the infinite list to build a new list tup consisting of tuples. The tuples must look as follows:

   [(1,1/1),(3,1/3),(5,1/5),(7,1/7),...,(i,1/i), ...

3. Here is piece of code.

   ```
   plop x = plop x

   quango u v = u ++ (if (length u) > 3 then "ringo" else v)
   ```

   Explain as precisely as possible *why* the call

   ```
   quango "paul" (plop 14)
   ```

   succeeds. Write your answer as a comment in solutions.hs.

Problem 6 – 16 points

A frequency table ft is a finite list of type Integral b => (a,b) of the form

$$[(x_1, t_1), \ldots (x_k, f_k)]$$

where $f_1, \ldots, f_k$ are positive whole numbers. The intention is that a frequency table ft can tell us how many times elements occur in a list. We say that x is *mentioned* in ft if there is a pair (x, f) somewhere in ft.

In the following, we assume that every frequency table ft is *well-formed*: if x is mentioned in ft, it is mentioned exactly once, and then (x, f) has $f > 0$.

We say that a well-formed frequency table ft $= [(x_1, t_1), \ldots (x_k, f_k)]$ is *valid* for a list xs if the table correctly tells us how many times each element occurs in xs and tells us nothing more. In other words, for every element x of xs, x is mentioned in ft with the pair (x, f) and there are exactly f occurrences of x in xs. Moreover, ft does not mention elements that are not found in xs.

As an example, ft1 = [("a",2),("b",1)] is valid for the list ["a","b","a"]. On the other hand, we have that ft2 = [("a",1),("b",1)] is not valid for the list ["b","a","b"] (because "b" occurs two times in the list) or for the list ["a","b","b","c"] (because "c" is not mentioned in ft2). Moreover, ft2 = [("a",1),("b",1)] is not valid for the list ["b"] (because ft2 mentions "a" which does not occur in ["b"]).

1. Define a function dec that takes a frequency table ft (which we assume is well-formed; this is not supposed to be checked by dec) and an element x and gives us a new frequency table in which the occurrence count of x is decreased by 1 if x was mentioned in ft as (x, f) with $f > 1$ and removes x from ft if $f = 1$. Otherwise, ft is returned.

   So dec [("a",2),("b",1)] "a" should give us [("a",1),("b",1)]. And dec [("a",2),("b",1)] "b" should give us [("a",2)].

2. Define a function validfq that will tell us if a frequency table (which we assume is well-formed; this is not supposed to be checked by validfq) is valid for a list. What should its type be?