Programming Paradigms
Written Examination

Aalborg University

Practice Exam
12:45 - 15:45

Read the following very carefully before you begin

What is this? This exam paper consists of 6 questions. The paper is part of a zip file along with the compendium and a file called `solutions.hs`.

How do I start? *You must do the following immediately!!* Unzip the zip file – it is not enough to just look inside; you *must* unzip it and save the exam paper and `solutions.hs` locally on your computer. Otherwise, the answers you write in `solutions.hs` will not be saved. Write your full name, your AAU email address, and your student number at the top of your local copy of `solutions.hs`, where indicated in the file. **Do it immediately.** If you experience issues with the file type `.hs`, rename the file while working and restore the `.hs` extension just before submitting it.

How and where should I write my solution? You must write your answers in the local copy of `solutions.hs` on your computer. You must indicate *in comments* which question your text pertains to and which sub-question it addresses. Any other text that is not valid Haskell code (such as code containing syntax errors or type errors) must also be written as comments. Use the format shown in the example below.

```
-- Question 2.2
bingo = 17
-- The solution is to declare a variable called bingo with the value 17.
-- Here is a test.
-- ghci> bingo
-- ghci> 17
```

Your file must be compilable. Ensure this is the case before submitting your solution. Comment out any parts that prevent the file from compiling.

How should I submit my solution? This is a practice exam, so in this case, you should submit the file via Moodle. *Submit the local copy of `solutions.hs` containing your solutions, and nothing else.* **DO NOT SUBMIT A ZIP FILE.**

What am I allowed to consult during the exam? During the exam, you may only use the textbook *Programming in Haskell* by Graham Hutton, the compendium, and your installation of the Haskell programming environment. **Nothing else is permitted.** Access to Moodle, except for downloading this file and submitting your solution, is also not allowed.

Which libraries am I allowed to use for my code? You may only use the Haskell `Prelude` for your Haskell code, unless the text of a specific question explicitly states that you should use a particular Haskell library. Do not use any special GHCi directives.

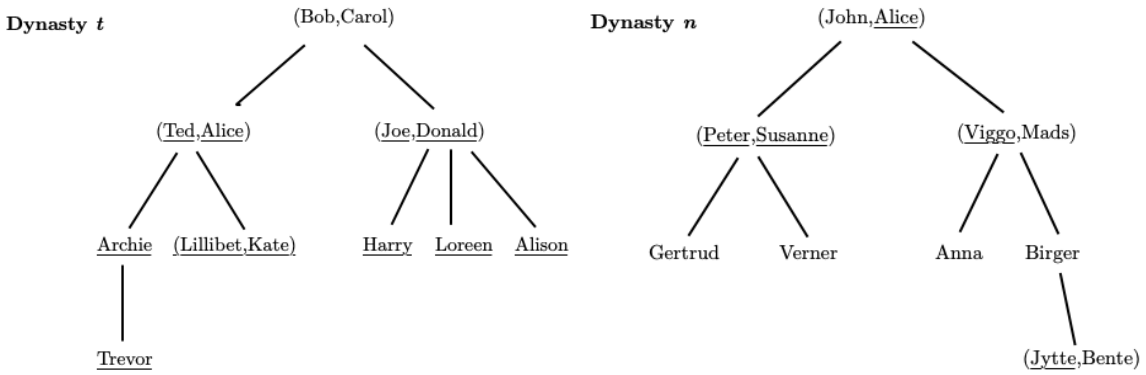

Figure 1: A modern dynasty t and a dynasty n that is not modern

Question 1 – 18 points

1. Provide a *recursive* definition of a function remove, which takes a list xs and an index k, and returns the list with the element at index k removed. Assume that elements in a list are numbered from 1 upwards. If k is less than 1 or greater than the length of the list, return xs. We expect remove [17,42,484000,9400] 2 to give us [17,484000,9400], and remove "mango" 7 to give us "mango". Is the function polymorphic? If so, is the polymorphism parametric, ad hoc, or both? Justify your answer. Provide three test cases and show the results from GHCi for your tests in a comment. *Solutions that do not use recursion will receive no points.*

2. Use the function remove to define a function removals, which takes a list xs as an argument and returns a list of lists, where the ith list is xs with the ith element removed. For example, removals [1,2,3,4] should give us

   ```
   [[2,3,4],[1,3,4],[1,2,4],[1,2,3]]
   ```

   Provide three test cases and show the results from GHCi for your tests in a comment.

Question 2 – 20 points

In Tabloidland, every person is either famous or ordinary and has a name, which is a text string. A *family* in Tabloidland can be a single person or a couple consisting of two people. A family can have an arbitrary number of children (including none). A family is famous if it is either a single person x who is famous, or a couple (x, y) where both x and y are famous. All other families are ordinary. A *dynasty* is a tree of families, and its root is called the dynasty's *head*. Figure 1 shows two dynasties in Tabloidland. The famous people in the dynasty are shown with underlined names. In a couple, the child of the family above always comes first in the pair, so the couple (Viggo,Mads) has the children Anna and Birger. A dynasty is *modern* if the following two conditions are both satisfied:

- If a family in the dynasty is famous, then every child of the family is also famous.
- Any family (single or couple) that includes a child of a famous family is famous and is the head of a modern dynasty.

Thus, a dynasty is modern if either no families in the dynasty are famous or if all families are famous from a certain level onward.

The dynasty t is modern. The head (Bob,Carol) is not famous, but both their children Ted and Joe are heads of a famous dynasty. The dynasty n is not modern. Here, the head of the couple is (Peter,Susanne). Peter and Susanne are famous, but neither Gertrud nor Verner is famous.

1. Define a datatype Dynasty that describes a dynasty. Show how the dynasties t and n in Figure 1 can be represented using your datatype.

2. Define a function modern with the type modern :: Dynasty -> Bool, such that modern d evaluates to True if d is a modern dynasty, and evaluates to False otherwise. modern n should return False, and modern t should return True.

   Provide three test cases and show the results from GHCi for your tests in a comment.

Question 3 – 18 points

Define a function dwindle, which takes a list xs. If the list is not empty, dwindle xs produces a list of all non-empty suffixes of the list, ending with the list consisting of the last element of the list. If the list is empty, dwindle xs produces [[]]. For example, dwindle "mango" should give us ["mango","ango","ngo", "go","o"], and dwindle [6,3,0,1,2,5] should give us [[6,3,0,1,2,5], [3,0,1,2,5], [0,1,2,5], [1,2,5], [2,5],[5],[]].

1. Define dwindle using recursion without using list comprehension.
2. Define dwindle using list comprehension but without using recursion.
3. Define dwindle using foldr or foldl *without using* recursion or list comprehension in any way.

Provide three test cases and show the results from GHCi for your tests in a comment.

Question 4 – 18 points

Here is the declaration of a type Status and a declaration that makes it an instance of Functor. The idea is that we now classify data as either used or fresh.

```
data Status a = Fresh a | Used a deriving Show
instance Functor Status where
   fmap f (Fresh x) = Fresh (f x)
   fmap f (Used x) = Used (f x)
```

1. Extend the above code with an instance declaration so that Status also becomes an applicative functor. Elements that we wrap in this type should become fresh.
2. Extend the above code with an instance declaration so that Status also becomes a monad.
3. Use a do-block in the Status monad, which you now have, to define a function

   ```
   minimise :: Ord b => Status b -> Status b -> Status b
   ```

   which, given two values of type Status b, where b is a type in Ord, gives us the smallest value, which is then set to be fresh. minimise (Fresh 4) (Fresh 8) should give us Fresh 4, and minimise (Used "ab") (Used "aa") should give us Fresh "aa". Provide three test cases and show the results from GHCi for your tests in a comment.

Question 5 – 8 points

Here are five types. For each of the five types, you must:

- Find an expression or function definition in Haskell that has this specific type.
- Explain whether polymorphism is involved, and if so, whether it is parametric polymorphism, ad hoc polymorphism, or a mixture of both.

Write your explanations in comments. Definitions should not be in comments.

1. (Num a => Bool -> a -> (a, a)
2. a1 -> a2 -> (a2, a2) -> (a1, [a2])
3. Num a => Maybe [a -> a]
4. p1 -> p2 -> [a]
5. Bool -> String

Question 6 – 18 points

1. Use recursion to provide a definition of the list squares of square numbers. 1 is the smallest square number.

2. Use map to provide an alternative definition of squares called squares'. We thus expect that take 10 squares' is

   ```
   [1,4,9,16,25,36,49,64,81,100]
   ```

3. Here is a definition of a value v.

   ```
   v  = (\y -> (tail [1 / (3-y), 7.9, 11.3])) (1+2)
   ```

   Provide a reduction sequence (as in Chapter 15 of *Programming in Haskell*) showing what happens when v is evaluated. Justify your answer as precisely as possible.