Programming Paradigms
Written exam

Aalborg University

13 January 2023
9:00 - 12:00 am

You must read the following before you begin

1. This problem set consists of 6 problems. The problem set is part of a zip archive together with a file `solutions.hs`.
2. Please add your full name, AAU mail address and study number to the top of `solutions.hs` where indicated in the file. This should be the very first thing that you do.
3. You can write your answers in Danish or in English.
4. Please write your answers by adding them to `solutions.hs`. You must indicate in comments which problem your text concerns and which subproblem it concerns. All other text that is not runnable Haskell code 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.
   ```

5. Submit the resulting version of `solutions.hs` and nothing else to Digital Eksamen when you are done.
6. During the exam, you are allowed to use the textbook *Programming in Haskell* by Graham Hutton, your own notes and your installation of the Haskell programming environment.
7. 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. No special GHCi directives are to be used.
8. Please read the text of each problem carefully before trying to solve it. All the information you will need is in the problem text itself.

Problem 1 – 16 points

The goal of this problem is to define a function partition which takes a predicate, p, and a list lst, and returns a pair of lists (yes,no) such that the list yes contains the elements of lst for which p is true and the list no contains the elements of lst for which p is false. In both yes and no the elements appear in the same order as they did in lst.

Here are three examples of what partition will do.

partition (odd) [1,2,3,4,5] gives us ([1,3,5],[2,4])

partition (== "la") ["tra","la","la","tra","la"] gives us (["la","la","la"],["tra","tra"])

partition (> False) [True,False,False,True] gives us ([True,True],[False,False])

1. What is the type of partition? Is the partition function polymorphic? If yes, explain if partition uses parametric polymorphism or ad hoc polymorphism (overloading) or a combination of them and why. If no, explain why partition is not polymorphic.

2. Give a definition of partition in Haskell that uses list comprehension only.

3. Give a definition in Haskell of partition that uses that uses recursion but not list comprehension or higher-order functions.

Problem 2 – 18 points

This goal of this problem is to define a data structure and a function over it.

We consider a data structure that we call mixed a, b-trees. Let a and b be any types. Then we say that a tree is a mixed a, b-tree if it is has one of these forms given below.

- It can be an empty leaf
- It can be a leaf labelled with an element of type a
- It can be a leaf labelled with an element of type b
- It can consist of a node labelled with an element of type a and a left and right subtree that are also mixed a, b-trees.
- It can consist of a node labelled with an element of type b and a left and right subtree that are also mixed a, b-trees.

Figure 1 shows two mixed Int, String-trees; empty leaves are not shown.

(a) A mixed Int, String-tree that is layered (b) A mixed Int, String-tree that is not layered

Figure 1: Two mixed Int, String-trees

1. Define a datatype MixedTree for mixed a, b-trees in Haskell.
2. Represent the tree in Figure 1a as a term of type MixedTree.

We say that a mixed a, b-tree t is *layered* if the following holds:

- Every node in t labelled with an element of type a has all its children labelled with elements of type b.
- Every node in t labelled with an element of type b has all its children labelled with elements of type a.

The tree in Figure 1a is layered, the tree in Figure 1b is not.

3. Define a function islayered which takes a tree as argument and tells us if the tree is layered.

Problem 3 – 16 points

The goal of this problem is to define values, given specifications of them in the form of types.

Below are four types. For each of them, define a Haskell value (which may be a function) that has this particular type.

1. `(Eq a, Num a) => a -> a -> [b] -> (b, b)`
2. `(Show a, Fractional a) => p -> a -> [Char]`
3. `[a] -> [a] -> [a]`
4. `(t1 -> t2) -> ((b, b) -> t1) -> b -> t2`

For each of these four types indicate if the type involves

- parametric polymorphism only
- overloading (ad hoc-polymorphism) only
- both forms of polymorphism

Problem 4 – 16 points

The goal of this problem is to define a function lists that will generate an infinite list.

When given a natural number n, lists n will generate an infinite list of lists of natural numbers in which every number n occurs exactly n times.

First a list of n copies of n, then a list of $n+1$ copies of $n+1$, then a list of $n+2$ copies of $n+2$ and so on.

As an example, a call to lists 4 will result in an infinite list beginning with the following fragment

[[4,4,4,4],[5,5,5,5,5],[6,6,6,6,6,6],[7,7,7,7,7,7,7],[8,8,8,8,8,8,8,8]...

1. Is lists a polymorphic function? If yes, explain if lists uses parametric polymorphism or ad hoc polymorphism (overloading) or a combination of them and why. If no, explain why lists is not polymorphic.

2. Define lists by a definition that uses recursion but not list comprehension or higher-order functions.

3. Define lists without recursion by using higher-order functions (such as map, foldr) and possibly also list comprehension.

Problem 5 – 18 points

The goal of this function is to define a functor instance for a type and prove a property of the definition that you make.

Here is a type declaration.

```
newtype Funpair a = Fun (Bool -> a, String -> a)
```

1. Give an example of an expression in Haskell that has type Funpair Integer.
2. Show how to define Funpair to be a functor.
3. Show that the functor law

   $$\text{fmap id} = \text{id}$$

   holds for your definition.

Problem 6 – 16 points

The goal of this function is to complete the definition of a monad and define two functions that involve this monad.

Below is a piece of Haskell code.

```
data W x = Bingo x deriving Show

instance Functor W where
  fmap f (Bingo x) = Bingo (f x)

instance Monad W where
  return x = Bingo x
  Bingo x >>= f = f x
```

1. For this to make sense, a definition of W as an applicative functor is missing. Write such a definition.

2. Use do-notation to define a function `wrapadd :: Int -> W Int -> W Int` which satisfies that

   $$\texttt{wrapadd x (Bingo y)} = \texttt{Bingo (x+y)}$$

3. Use do-notation to define a function `h :: W Int -> W Int -> W Int` which satisfies that

   $$\texttt{h (Bingo x) (Bingo y)} = \texttt{Bingo (x*y)}$$