Programming Paradigms 2025
Session 11: Monads

Problems for solving and discussing

Hans Hüttel

18 November 2025

Problems that we will definitely talk about

1. ***(10 minutes)***

   Two influencers were having a heated discussion about the List monad. Influencer 1 presented a new function called fourfirst :

   ```
   fourfirst xs = do
                    x <- xs
                    return (4,x)
   ```

   "This function takes a list and gives us a pair $(4, x)$ where x is the first element of the list", the first influencer said.

   "You are wrong", said Influencer 2. "The code assigns xs to the variable x and we get a pair where the first component is 4 and the second component is x, which is a list that is also known as xs."

   Explain, using the definition of the List monad *but without executing the code*, what this actually does and why.

2. ***(25 minutes)***

   Here 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
   ```

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

   b. A major clothing company sponsors a popular TV show and asked the star of the show to define a function wrapadd :: Num b =>b -> W b -> W b which satisfies that

   ```
   wrapadd (Bingo x)(Bingo y) = Bingo (x+y)
   ```

   and to make use of the fact that W is monad. The definition was to be used in advertisements on social media for a series of new jackets.

   The TV star came up with the definition

   ```
   wrapadd x (Bingo y) = do
                           Bingo (x+y)
   ```

   However, the clothing company complained that this definition was clumsy – it did not use monads in a sensible way. The TV star was asked to revise the definition. What is wrong with it?

```
data Aexp = Var String | Num Integer | Plus Aexp Aexp | Mult Aexp Aexp

look ass x = head [ v | (y,v) <- ass, y == x ]

eval (Var x) ass = look ass x
eval (Num n) ass = n
eval (Plus a1 a2) ass = v1 + v2
                        where
                          v1 = eval a1 ass
                          v2 = eval a2 ass
eval (Mult a1 a2) ass = v1 * v2
                        where
                          v1 = eval a1 ass
                          v2 = eval a2 ass

-- An example assignment
myass = [("x",3),("y",4)]
```

Figure 1: The original solution that defines an evaluator

3. ***(30 minutes)***

   In session 7 we looked at a Haskell datatype Aexp for arithmetic expressions with addition, multiplication, numerals and variables. The formation rules are

   $$E ::= n \mid x \mid E_1 + E_2 \mid E_1 \cdot E_2$$

   We assume that variables x are strings and that numerals n are integers.

   We were asked to define a function eval that, when given a term of type Aexp and a value assignment ass of variables to numbers, will compute the value of the expression.

   As an example, if we have the assignment $[x \mapsto 3, y \mapsto 4]$, eval should tell us that the value of $2 \cdot x + y$ is 10.

   Figure 1 shows the code for a solution to this problem.

   However, the above definition cannot deal with cases in which we try to evaluate an expression that mentions variables whose value is given in the assignment that we supply.

   As an example, if we try to evaluate the expression $2 + z$ with the assignment $[x \mapsto 3, y \mapsto 4]$, an error will occur.

   Your task is now to *use the Maybe monad* to define a new evaluator, eval' such that eval' e ass will return Just v if e evaluates to the value v and return Nothing if e mentions variables that are not bound in ass.

   Find at least four good test cases.

More problems to solve at your own pace

a) Define a foldM function whose type should be

```
foldM :: Monad m => (t1 -> t2 -> m t2) -> [t1] -> t2 -> m t2
```

The idea is that the function works like foldl but folds over a monad.

Here is an example that shows what will happen if we fold over the IO monad. If we let

```
dingo x = do
            putStrLn (show x)
            return x
```

then we should see the following behaviour.

```
*Main> foldM (\x y -> (dingo (x+y))) [1,2,3,4] 0
1
3
6
10
```

b) Consider trees whose elements are values of some type in the type class Ord. The type Tree a is defined by

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

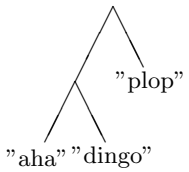

Figure 2: The ordered mytree1

Use a monad to write a function minorder that takes such a tree and checks if the numbers in the structure are in non-decreasing order when read from left to right. If it is, the function should return Just k, where k is the smallest number in the tree, otherwise it should return Nothing. The tree mytree shown in Figure 2 is ordered, so minorder mytree1 should return Just "aha'', but the tree in Figure 3 is not ordered, so minorder mytree2 should return Nothing.

First define another function minmax that finds the minimal and the maximal element in a tree under the assumption that the tree is ordered. Then use minmax to define minorder.

Hints: First, find some good test cases. Then find out which monad you should use.

Warning! Only use monadic notation!

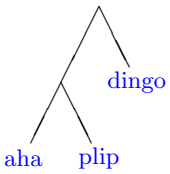

Figure 3: The unordered mytree2