Programming Paradigms 2022
Session 10: Monads

Problems for solving and discussing

Hans Hüttel

15 November 2022

Problems that we will definitely talk about

1. ***(Work in pairs – 15 minutes)***

   A famous influencer writes frequent updates to Instagram about the List monad. Yesterday the influencer 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 influencer concluded. Soon after a heated discussion had begun among 4 million teenage followers: Was the influencer right? It is your task to find out. Explain, using the definition of the List monad *but without executing this piece of code*, what the code actually does and how it does it.

2. ***(Everyone at the table together – 15 minutes)***

   Let us change the notion of state from pages 168 and onwards such that a state is now a stack, modelled as a list of integers. A state is now of the form SE xs, where SE is the term constructor for states and xs is a list of integers.

   The top element of the stack (if it exists) is the head of the list. You can find the revised definitions in the file `state.hs` on the course Moodle in the entry for this session or copy them from here:

   ```
   newtype State = SE [Int] deriving Show

   newtype ST a = ST (State -> (a, State))

   app :: ST a -> State -> (a,State)

   app (ST st) x = st x
   ```

   Remember that the type ST a is the type of a state transformer that given a state returns a value of type a and a new state. The difference is that states are now lists of integers.

   The goal of this problem is to implement three stack operations. Their types are

   ```
   get :: ST [Int]
   put :: put :: Int -> ST Int
   remove :: ST Int
   ```

   The stack operations are intended to work as follows.

   - get is a state transformer that, given a state s, gives us the top of the stack, if it exists, and 0 otherwise *and* the same state s
   - put takes an integer x as its argument and then, given a state s, gives us a state s' where x has been placed on top of the stack
   - remove takes a state s and gives us a state s' where the top element from the stack of s has been discarded

   Define the get operation in Haskell. *You do not need monads for this problem. This is an exercise in using term constructors and pattern matching.*

3. ***(Work in pairs)* – 15 minutes**

   Now define the put and remove operations in Haskell. *You still do not need monads.*

4. ***(Everyone at the table– 40 minutes))***

   The instance declarations that make the ST type constructor a monad are exactly as on pages 159-160. You can find them in the file `state.hs` on the course Moodle in the entry for this session.

   The goal of this problem is to define a small stack machine language and then to show you can use these functions to write programs for the stack machine using the ST monad.

   The four operations of the stack machine are four functions that implement the following instructions of a simple stack machine:

   - push that gives us a stack where a new integer is placed on top of the stack
   - pop that gives us the top element of a stack and a stack from which this top element has been removed
   - add that addes the two uppermost values x and y on the stack gives us a stack with a new top element, $x + y$, and where x and y have been removed
   - mult that addes the two uppermost values x and y on the stack gives us a stack with a new top element, $x \cdot y$, and where x and y have been removed

   Before you try to define these functions, figure out what their types should be. Here it is a good idea to recall what a state transformer is and how you can express a state transformer in Haskell.

   Then use the definitions of the functions from the previous problem to implement these four new stack machine operations.

   When you have defined the functions find out, using the ST monad, how to express the stack machine program

   ```
   push 2
   push 3
   add
   push 5
   mult
   pop
   ```

   using a do block. Finally find out by evaluating the code using Haskell using the app function what the result of the stack machine program will be when given a start state SE [0]. This start state corresponds to a stack that contains only the integer 0.

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
   10
   ```

   where the last number appears twice, as it is the final result.