Programming Paradigms 2024
Session 11 : Monads

Hans Hüttel

26 November 2024

Our plan for today

1. The learning goals
2. Presentations of the preparation problems
3. Problem no. 1
4. How *not* to use monads
5. Problem no. 2
6. Break
7. A bedtime story
8. Problem no. 3
9. Problem no. 4
10. If time allows: More problems at your own pace.
11. We evaluate today's session – please stay until the end!
12. Everyone: The semester evaluation

Learning goals

- To understand the Maybe monad
- To understand the definition of a monad
- To understand the *bind* operator (>>=) and how it can be expressed using do notation
- To understand the definition of the List monad
- To understand the definition of the State monad and how it can be used in programming with state

Preparation problem – tuple

Define a function

```
tuple :: Monad m => m a -> m b -> m (a, b)
```

using explicit (>>=) and then again, now using do-notation. What does the function do in the case, where the monad is Maybe?

Preparation problem – do-notation and (>>=)

What is the expression (that uses (>>=) equivalent to the following do block?

```
do y <- z
   s y
   return (f y)
```

Discussion / Problem 1 – 15 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", said the first influencer concluded.

"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."

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

Problem 2 – Bingo (10 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 definition of W as an applicative functor is missing. Write such a definition.

Discussion – How not to use monads

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

`h (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.

As an example, the clothing company would like `wrapadd (Bingo 5)(Bingo 3)` to return `(Bingo 15)`.

The TV star came up with the definition

```
wrapadd (Bingo x) (Bingo y) = do
                  return (x*y)
```

However, the clothing company complained that this definition did not use monads in a sensible way. Why?

Break

Problem 3 – Addition in Bingo (15 minutes)

The TV star was asked to revise the definition.

Write a sensible definition of wrapadd which makes use of the fact that Bingo is a monad.

Discussion – A bedtime story

A long time ago at a university far, far away an influencer was up late. "Tomorrow I have an exam. I will be writing my first-ever Haskell program. Maybe I ought to prepare?" the influencer wondered and found a PDF version of the textbook at very low cost. In it, the influencer saw the following piece of code:

```
mapM f [] = return []
mapM f (x:xs) = do
                y <- f x
                ys <- mapM f xs
                return (y:ys)
```

"I knew it! Haskell has *assignments*. Functions use return to return a value. There are *arrays*, too. This is just C all over again. " the influencer thought and went straight to bed. Was the influencer right? What does this piece of code do?

Problem 4 – Trees (30 minutes)

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)
```

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 the smallest number in the tree, otherwise it 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. Maybe there is a good answer.

Evaluation

- What did you find difficult?
- What surprised you?
- What went well?
- What could be improved? How does the setup work this time?
- Is there a particular problem that we should follow up on with a short video?