Functional Programming - 202001186

Monads

Pietro Pennestrì (s2382660), Adhithya Ramamoorthy (s2363984)

>>= Contents

In this presentation the following topics are covered

1. Monads concept is introduced by means of an example;
2. Solutions details of the Tape assignment.

An Introductory Example I

Let us consider the data type `Expr`

```haskell
data Expr = Val Int | Div Expr Expr
```

and the corresponding evaluation function:

```haskell
eval :: Expr -> Int
eval (Val n) = n
eval (Div x y) = (eval x) `div` (eval y)
```

An Introductory Example II

- The `eval` function encounters an error when division by zero occurs.
- To handle the division by zero case, the `eval` function is redefined as follows.

An Introductory Example III

```haskell
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv x y = Just (x `div` y)
```

```haskell
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) =
    case (eval x) of
        Nothing -> Nothing
        Just n -> case (eval y) of
                      Nothing -> Nothing
                      Just m -> safediv n m
```

An Introductory Example IV

The new definition of `eval` function:

- will never crash;
- is more complex than the original one.

Is it possible to simplify the definition of the `eval` *function* by using an applicative style?

An Introductory Example V

```
1 eval :: Expr -> Maybe Int
2 eval (Val n) = pure n
3 eval (Div x y) = pure safediv <*> (eval x) <*> (eval y)
```

Wrong!

Required type	`Int -> Int -> Int`
Provided type	`Int -> Int -> Maybe Int`

It is impossible to use an applicative style to define the `eval` function.

An Introductory Example VI

- The common pattern used in the evaluation of `eval` is:

```
1 case ____ of
2     Nothing -> Nothing
3     Just x -> ____ x
```

- *Is it possible to turn the previous pattern into a definition?*

The Bind Operator »=

- It is possible to define a bind operator »= :

```
1 mx >>=: f =
2     case mx of of
3         Nothing -> Nothing
4         Just x -> f x
```

- The bind operator has the following type:

```
1 (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Warning this is not the general definition of »=

The `eval` function definition using the bind operator »=

- It is possible to redefine the `eval` function by means of the »= operator

```
1 eval :: Expr -> Maybe Int
2 eval (Val n) = Just n
3 eval (Div x y) = eval x >>= ( \n ->
4     eval y >>= ( \m ->
5         safediv n m
6     )
7 )
```

- The failure management is handled by the binding machinery.

>>=

Monad Class definition

The Monad concept is described in `Haskell` with the following built-in declaration:

```haskell
class Applicative m => Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  return = pure
```

A Formal Definition of Monads

From the previous built in class definition the following `Monad` definition arises:

Definiton

A `Monad` is an applicative type m that support `>>=` and `return` functions of the specified type.

The `do` Notation

```
1 m1 >>= \x1 ->
2 m2 >>= \x2 ->
3
4
5 mn >>= \xn ->
6  f x1 x2        xn
```

```
1 do x1 <- m1
2   ; x2 <- m2
3
4
5   ; f x1       xn
```

Quick Introduction

The assignment deals with a computational unit equipped with a tape. The tape has the followings characteristics:

- The tape head is located at a position, and we can apply functions to the value at its current position by reading to / writing from the tape.
- It is possible to move one location backward/forward or rewind the tape.

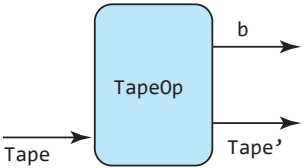
Data Types

- The Tape (the state)

```haskell
data Tape a = Tape (Int, [a])
```

- A Tape operation (state transformer) (`TapeOp`)

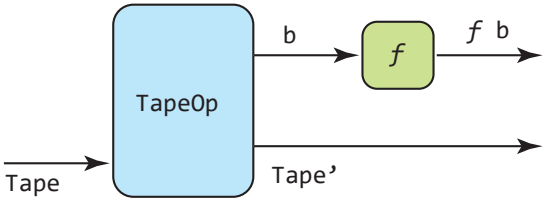
```haskell
data TapeOp a b = TapeOp {
 exec :: Tape a -> (Tape a , b)
}
```



`Functor` instance of `TapeOp`

```haskell
instance Functor (TapeOp a) where
  -- fmap (b->c) -> TapeOp a b -> TapeOp a c
  fmap f st =  TapeOp (\s -> let(s',x) = exec st s in (s' , f x) )
```

`Functor` instance of `TapeOp`: A graphical representation

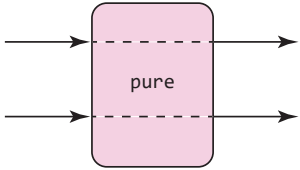


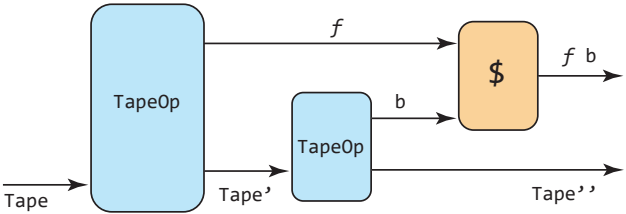
`fmap` allows us to apply a function to the result value of a tape operation

`Applicative` instance of `TapeOp`

```haskell
instance Applicative (TapeOp a) where
  -- pure :: b -> TapeOp a b
  pure x = TapeOp (\s -> (s,x) )
  -- (<*>) :: TapeOp a (b->c) -> TapeOp a b -> TapeOp a c
  stf <*> stx = TapeOp ( \s ->
   let (s',f)   = exec stf s
       (s'', x) = exec stx s' in (s'', f x) )
```

`Applicative` instance of `TapeOp`: A graphical representation





A Remark on `pure` Function

- In this case `pure` transforms the provided input *i* into a `TapeOp` (state transformer) that simply returns *i* without modifying the tape (state).

Example:

```
*Tape> mytape
Tape (0,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14])
*Tape> a = pure 5 :: TapeOp a Int
*Tape> exec a mytape
(Tape (0,[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14]),5)
*Tape>
```

Monad instance of TapeOp

```haskell
instance Monad (TapeOp a) where
 -- return :: b -> TapeOp a b
 return x = TapeOp (\s -> (s,x) )  -- return is the same as pure
 -- (>>=) :: (TapeOp a b) -> (b -> TapeOp a c) -> TapeOp a c
 st >>= f = TapeOp (\s -> let (s',x) = exec st s in exec (f x) s' )
```

»= for `TapeOp` a Step by Step Explanation

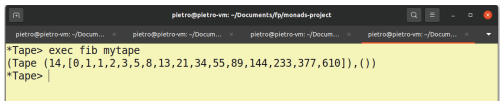
`st »= f`:

1. executes the tape operation `st` on the tape `s` which returns an update version of the tape `s'` and a value `x`;
2. applies the function `f` to `x`, that results into a tape operation `st'`
3. executes the tape operation `st'` on the tape `s'`.

Testing of the `Monad` instance for `TapeOp`

The `Monad` instance for `TapeOp` has been verified with the following function:

```
1  ----------------------
2  -- Assignment - TAPE4
3  ----------------------
4  -- run the fib function: exec fib mytape
5  fib :: TapeOp Integer ()
6  fib = do rewind
7           wr 0
8           right
9           wr 1
10          left
11          n <- tapelen
12          replicateM (n-1) (do x <- rd
13                               right
14                               y <- rd
15                               right
16                               wr (x+y)
17                               left)
18          pure ()
```



Conclusions

- This project allowed us to get familiar with the `Monad` concept;
- The exercises gave us the opportunity of testing the theoretical knowledge gathered during the course.

References

- Programming in Haskell (2nd ed.), G. Hutton
- What is a Monad? - Computerphile
  `https://www.youtube.com/watch?v=t1e8gqXLbsU`