Curs 2

Cuprins

Haskell: Clasa de tipuri **Monad**

Clasa de tipuri **Monad**

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

    ma >> mb = ma >>= \_ -> mb
```

- m a este tipul computațiilor care produc rezultate de tip a (și au efecte laterale)
- a -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor

Functor și Applicative definiți cu **return** și >>=

```
instance Monad M where
  return a = ...
  ma >>= k = ...

instance Applicative M where
  pure = return
  mf <*> ma = do
    f <- mf
    a <- ma
    return (f a)
   -- mf >>= (\f -> ma >>= (\a -> return (f a)))

instance Functor F where          -- ma >>= \a -> return (f a)
  fmap f ma = pure f <*> ma       -- ma >>= (return . f)
```

Notația **do** pentru monade

Notația cu operatori	Notația **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

Notația **do** pentru monade

Notația cu operatori	Notația **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

De exemplu

```
e1    >>=  \x1  ->
e2    >>  e3
```

devine

Notația **do** pentru monade

Notația cu operatori	Notația **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

De exemplu

```
e1    >>= \x1 ->
e2    >> e3
```

devine

```
do
    x1 <- e1
    e2
    e3
```

Exemple de efecte laterale

I/O	Monada **IO**
Parțialitate	Monada **Maybe**
Excepții	Monada **Either**
Nedeterminism	Monada [] (listă)
Logging	Monada Writer
Memorie read-only	Monada Reader
Stare	Monada State

Implementări pentru intrări/ieșiri

În continuare vom implementa propria monadă **IO**.

```
type Input = String
type Output = String


newtype MyIO a =
        MyIO { runMyIO :: Input -> (a, Input, Output) }

instance Monad MyIO where
      ...
```

O data myio :: MyIO a are forma myio = MyIO f unde
f :: Input -> (a, Input, Output) și runMyIO myio = f

Monada MyIO

```
instance Monad MyIO where
    return x = MyIO (\input -> (x, input, ""))
    m >>= k  = MyIO f
            where f input =
                    let (x, inputx, outputx) = runMyIO m input
                        (y, inputy, outputy) = runMyIO (k x)
                                               inputx
                    in  (y, inputy, outputx ++ outputy)

instance Applicative MyIO where
    pure      = return
    mf <*> ma = do { f <- mf; a <- ma; return (f a) }

instance Functor MyIO where
    fmap f ma = do { a <- ma; return (f a) }
```

MyIO - funcționalități de bază

```
newtype MyIO a =
  MyIO { runMyIO :: Input -> (a, Input, Output) }

myPutChar :: Char -> MyIO ()
myPutChar c = MyIO (\input -> ((), input, [c]))

myGetChar :: MyIO Char
myGetChar = MyIO (\ (c:input) -> (c, input, ""))

runIO :: MyIO () -> String -> String
runIO command input = third (runMyIO command input)
                      where third (_, _, x) = x
-- primind o  comanda si un sir de intrare, intoarce sirul
    de iesire
```

MyIO - myGetChar și myPutChar

Exemple de utilizare:

```
> runMyIO myGetChar "abc"
('a',"bc","")

> runIO (myPutChar 'a' :: MyIO ()) ""
"a"

> runMyIO (myPutChar 'a' >> myPutChar 'b') ""
((),"","ab")

> runMyIO (myGetChar >>= myPutChar . toUpper) "abc"
((),"bc","A")
```

myPutStr folosind myPutChar

```
myPutStr :: String -> MyIO ()
myPutStr = foldr (>>) (return ()) . map myPutChar

myPutStrLn ::  String -> MyIO ()
myPutStrLn s = myPutStr s >> myPutChar '\n'
```

```
> runIO (myPutStr "abc" :: MyIO ()) ""
"abc"
```

myGetLine folosind myGetChar

```
myGetLine :: MyIO String
myGetLine = do
    x <- myGetChar
    if x == '\n'
      then return []
      else do
        xs <- myGetLine
        return (x:xs)

> runMyIO myGetLine "abc\ndef"
("abc","def","")
```

Exemple — Echoes

```
echo1 :: MyIO ()
echo1 = do {x<- myGetChar ; myPutChar x}

echo2 :: MyIO ()
echo2 =  do {x<- myGetLine ; myPutStrLn x}

> runMyIO echo1 "abc"
((),"bc","a")
> runMyIO echo2 "abc\n"
((),"","abc\n")
> runMyIO echo2  "abc\ndef\n"
((),"def\n","abc\n")
```

MyIO - exemplu

```
echo :: MyIO ()
echo = do
    line <- myGetLine
    if line == ""
      then return ()
      else do
        myPutStrLn (map toUpper line)
        echo
```

```
> runIO echo "abc\ndef\n\n"
 "ABC\nDEF\n"
```

Legătura cu **IO**

Vrem să folosim modalitățile uzuale de citire/scriere, adică să facem legătura cu monada **IO**. Pentru aceasta folosim funcția

interact :: (**String** –> **String**) –> **IO** ()

care face parte din biblioteca standard, și face următoarele:
- citește stream-ul de intrare la un șir de caractere (leneș)
- aplică funcția dată ca parametru acestui șir
- trimite șirul rezultat către stream-ul de ieșire (tot leneș)

```
convert :: MyIO () -> IO ()
convert = interact . runIO
```

Legătura cu **IO**

```
> convert echo
 aaa
 AAA
 bbb
 BBB
 ddd
 DDD
```

Monada MyIO

```
instance Monad MyIO where
    return x = MyIO (\input -> (x, input, ""))
    m >>= k  = MyIO f
            where f input
              let (x, inputx, outputx) = runMyIO m input
                  (y, inputy, outputy) = runMyIO (k x)
                      inputx
              in  (y, inputy, outputx ++ outputy)

instance Applicative MyIO where
    pure      = return
    mf <*> ma = do { f <- mf; a <- ma; return (f a) }

instance Functor MyIO where
    fmap f ma = do { a <- ma; return (f a) }

main :: IO ()
main = convert (echo :: MyIO ())
```

Clasa de tipuri pentru IO

Putem defini o clasă de tipuri pentru a oferi servicii de I/O

```
class Monad io => MyIOClass io where
    myGetChar :: io Char
    -- read a character

    myPutChar :: Char -> io ()
    -- write a character

    runIO     :: io () -> String -> String
    -- given a command and an input produce the output
```

Celelalte funcționalități I/O pot fi definite generic în clasa MyIOClass.

MyIO este instanță a lui MyIOClass

```
newtype MyIO a =
  MyIO { runMyIO :: Input -> (a, Input, Output) }

instance MyIOClass MyIO where
  myPutChar c = MyIO (\input -> ((), input, [c]))

  myGetChar = MyIO (\ (c:input) -> (c, input, ""))

  runIO command input = third (runMyIO command input)
    where third (_, _, x) = x
```

Pe săptămâna viitoare!

Pe săptămâna viitoare!