Programare declarativă - Implementări ale monadei `IO`

Traian Șerbănuță (33) Ioana Leuștean (34)

Departamentul de Informatică, FMI, UNIBUC
traian.serbanuta@fmi.unibuc.ro, ioana@fmi.unibuc.ro

Introducere

Implementări pentru intrări/ieșiri

```
import Data.Char (toUpper)
```

În acest curs vom prezenta două implementări ale monadei `IO`.

```
type Input = String
type Output = String
```

- O implementare folosește monada 'State:

  ```
  type InOutWorld = (Input, Output)
  type MySIO = State InOutWorld
  ```

- Pentru cealaltă implementare definim o monadă nouă:

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

  instance Monad MyIO where
    ...
  ```

Clasa de tipuri pentru IO

Un minim de functionalitate 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
```

Vom vedea că multe din celelalte funcționalități I/O pot fi definite generic în clasa MyIOClass.

Definirea MyIO folosind monada State

Monada State

```
newtype State state a
  = State { runState :: state -> (a, state) }

instance Monad (State state) where
  return a = State (\ s -> (a, s))
  ma >>= k = State g
    where g state =   let (a, aState) = runState ma state
                      in  runState (k a) aState

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

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

MySIO - definiție folosind monada State

```
type InOutWorld = (Input, Output)

type MySIO = State InOutWorld

-- MySIO a = State InOutWorld a
```

O data de tipul `myio :: MySIO a` are forma `myio = State f` unde `f :: InOutWorld -> (a, InOutWorld)` și `runState myio = f`

Instanța MyIOClass pentru MySIO

```
instance MyIOClass MySIO where
  myPutChar c
    = State (\(input, output) ->
                          ((), (input, output ++ [c])))

  myGetChar
    = State (\(c:input, output) -> (c, (input, output)))

  runIO command input
    = snd . snd $ runState command (input, "")
```

Observație: pentru definirea acestei instanțe e necesară activarea extensiei `FlexibleInstances`

```
`ghci -XFlexibleInstances myIO.lhs`
```

MySIO - getChar și putChar

Exemple de utilizare:

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

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

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

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

putStr folosind putChar

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

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

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

getLine folosind getChar

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

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

Exemple — Echoes

```
echo1 :: MyIOClass io => io ()
echo1 = do {x<- myGetChar ; myPutChar x}

echo2 :: MyIOClass io => io ()
echo2 =  do {x<- myGetLine ; myPutStrLn x}
```

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

MySIO - exemplu

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

```
> runIO (echo :: MySIO ()) "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 :: MyIOClass io => io () -> IO ()
convert = interact . runIO
```

Legătura cu IO

```
> convert echo
aaa
AAA

bbb
BBB

ddd
DDD
```

Definirea MyIO ca monada de sine stătătoare

Intrări/ieșiri fără State

Vom defini în continuare operațiile de intrare/ieșire folosind un tip de date nou.

```
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
```

Definițiile pentru myGetLine, myPutStr, echo, și convert merg și pentru MyIO

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 ())
```

Pe săptămâna viitoare!