

Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ana Cristina Țurlea

ana.turlea@fmi.unibuc.ro

- 1 Clasa de tipuri **Monad**
- 2 Monade standard
- 3 Monadă definită de utilizator: propria monadă IO

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 **compuțațiilor** 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 compuțațiilor

În Haskell, monada este o clasă de tipuri!

Functor și Applicative pot fi 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
```

```
    fmap f ma = pure f <*> ma
```

```
    -- ma >>= \a -> return (f a)
```

```
    -- ma >>= (return . f)
```

Notăția **do** pentru monade

Notăția cu operatori	Notăția do
$e \gg= \backslash x \rightarrow \text{rest}$	$x \leftarrow e$ rest
$e \gg= \backslash _ \rightarrow \text{rest}$	e rest
$e \gg \text{rest}$	e rest

De exemplu

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

devine

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

Monade standard

Exemple de efecte laterale

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

Monada **Maybe**(a funcțiilor parțiale)

```
data Maybe a = Nothing | Just a
```

```
instance Monad Maybe where
```

```
    return = Just
```

```
    Just va  >>= k    = k va
```

```
    Nothing >>= _    = Nothing
```

```
radical :: Float -> Maybe Float
```

```
radical x | x >= 0 = return (sqrt x)
```

```
          | x < 0  = Nothing
```

```
solEq2 :: Float -> Float -> Float -> Maybe Float
```

```
solEq2 0 0 0 = return 0                                --  $a * x^2 + b * x + c = 0$ 
```

```
solEq2 0 0 c = Nothing
```

```
solEq2 0 b c = return ((negate c) / b)
```

```
solEq2 a b c = do
```

```
    rDelta <- radical (b * b - 4 * a * c)
```

```
    return ((negate b + rDelta) / (2 * a))
```

Monada **Either**(a exceptiilor)

```
data Either err a = Left err | Right a
```

```
instance Monad (Either err) where
```

```
    return = Right
```

```
    Right va >>= k = k va
```

```
    err >>= _ = err    -- Left verr >>= _ = Left verr
```

```
radical :: Float -> Either String Float
```

```
radical x | x >= 0 = return (sqrt x)
```

```
          | x < 0  = Left "radical: argument negativ"
```

```
solEq2 :: Float -> Float -> Float -> Either String Float
```

```
solEq2 0 0 0 = return 0          --  $a * x^2 + b * x + c = 0$ 
```

```
solEq2 0 0 c = Left "Nu are solutii"
```

```
solEq2 0 b c = return ((negate c) / b)
```

```
solEq2 a b c = do
```

```
    rDelta <- radical (b * b - 4 * a * c)
```

```
    return ((negate b + rDelta) / (2 * a))
```

Monada listelor (a funcțiilor nedeterministe)

```
instance Monad [] where  
  return va = [va]  
  ma >>= k = [vb | va <- ma, vb <- k va]
```

Rezultatul funcției e lista tuturor valorilor posibile.

```
radical :: Float -> [Float]  
radical x | x >= 0 = [negate (sqrt x), sqrt x]  
            | x < 0  = []
```

```
solEq2 :: Float -> Float -> Float -> [Float]  
solEq2 0 0 c = [] --  $a * x^2 + b * x + c = 0$   
solEq2 0 b c = return ((negate c) / b)  
solEq2 a b c = do  
    rDelta <- radical (b * b - 4 * a * c)  
    return (negate b + rDelta) / (2 * a)
```

Monada Writer (variantă simplificată)

```
newtype Writer log a = Writer { runWriter :: (a, log) }  
-- a este parametru de tip
```

```
tell :: log -> Writer log ()  
tell msg = Writer ((), msg)
```

```
instance Monad (Writer String) where  
  return va = Writer (va, "")  
  ma >>= k = let (va, log1) = runWriter ma  
               (vb, log2) = runWriter (k va)  
               in Writer (vb, log1 ++ log2)
```

Monada Writer - Exemplu logging

```
newtype Writer log a = Writer { runWriter :: (a, log) }
```

```
tell :: log -> Writer log ()
```

```
tell msg = Writer ((), msg)
```

```
logIncrement :: Int -> Writer String Int
```

```
logIncrement x = do
```

```
    tell ("increment: " ++ show x ++ "\n")
```

```
    return (x + 1)
```

```
logIncrement2 :: Int -> Writer String Int
```

```
logIncrement2 x = do
```

```
    y <- logIncrement x
```

```
    logIncrement y
```

```
Main> runWriter (logIncrement2 13)  
(15,"increment: 13\nincrement: 14\n")
```

Monada State

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

```
instance Monad (State state) where
```

```
    return va = State (\s -> (va, s))
```

```
-- return a = State f where f = \s -> (a, s)
```

```
    ma >>= k = State $ \s -> let
```

```
        (va, news) = runState ma s
```

```
        State h = k va
```

```
    in (h news)
```

```
-- ma :: State state a
```

```
-- runState ma :: state -> (a, state)
```

```
-- k :: a -> State state b
```

```
-- h :: state -> (b, state)
```

```
-- ma >>= k :: State state b
```

Monada State

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

```
instance Monad (State state) where
```

```
  return va = State (\s -> (va, s))
```

```
-- return a = State f where f = \s -> (a, s)
```

```
  ma >=> k = State $ \s -> let
```

```
    (va, news) = runState ma s
```

```
    State h = k va
```

```
  in (h news)
```

Funcții ajutătoare:

```
get :: State state state
```

```
get = State (\s -> (s, s))
```

```
modify :: (state -> state) -> State state ()
```

```
modify f = State (\s -> ((), f s))
```

Monada State - exemplu "random"

```
newtype State state a = State{runState :: state ->(a, state)}  
get :: State state state  
get = State (\s -> (s,s))  
modify :: (state -> state) -> State state ()  
modify f = State (\s -> ((), f s))
```

```
cMULTIPLIER, cINCREMENT :: Word32  
cMULTIPLIER = 1664525 ; cINCREMENT = 1013904223
```

```
rnd, rnd2 :: State Word32 Word32  
rnd = do modify (\seed -> cMULTIPLIER * seed + cINCREMENT)  
      get  
rnd2 = do r1 <- rnd  
          r2 <- rnd  
          return (r1 + r2)
```

```
Main> runState rnd2 0  
(2210339985,1196435762)
```


Monada Reader(stare nemodificabilă)

```
newtype Reader env a = Reader { runReader :: env -> a }
```

```
ask :: Reader env env
```

```
ask = Reader id
```

```
instance Monad (Reader env) where
```

```
    return = Reader const    -- return x = Reader (\_ -> x)
```

```
    ma >>= k = Reader f
```

```
        where
```

```
            f env = let va = runReader ma env
```

```
                in  runReader (k va) env
```

Monada Reader- exemplu: mediu de evalua

```
newtype Reader env a = Reader { runReader :: env -> a }  
ask :: Reader env env  
ask = Reader id
```

```
data Prop ::= Var String | Prop :&: Prop  
type Env = [(String, Bool)]
```

```
var :: String -> Reader Env Bool  
var x = do  
    env <- ask  
    fromMaybe False (lookup x env)
```

```
eval :: Prop -> Reader Env Bool  
eval (Var x) = var x  
eval (p1 :&: p2) = do  
    b1 <- eval p1  
    b2 <- eval p2  
    return (b1 && b2)
```

Monada Writer(varianta lungă)

Clasele Semigrup și Monoid

Clasa de tipuri Semigroup

O mulțime, cu o operație $<>$ care ar trebui să fie asociativă

```
class Semigroup a where  
  (<>) :: a -> a -> a
```

Clasa de tipuri Monoid

Un semigrup cu unitatea mempty. mappend este alias pentru $<>$.

```
class Semigroup a => Monoid a where  
  mempty :: a  
  mappend :: a -> a -> a  
  mappend = (<>)
```

Foarte multe tipuri sunt instanțe ale lui Monoid. Exemplul clasic: listele.

Monada Writer(varianta lungă)

```
newtype Writer log a = Writer { runWriter :: (a, log) }
```

```
instance Monoid log => Monad (Writer log) where  
  return a = Writer (a, mempty)  
  ma >>= k = let (va, log1) = runWriter ma  
                (vb, log2) = runWriter (k va)  
                in Writer (vb, log1 `mappend` log2)
```

Monadă definită de utilizator: propria monadă IO

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", "")
```

Example — 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!