

Programare declarativă¹

Efecte laterale — Monade

Traian Florin Șerbănuță

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

6 ianuarie 2017

¹bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

Efecte laterale

Logging în C

```
#include <iostream>
#include <sstream>
using namespace std;
ostringstream log;
int increment(int x) {
    log << "Called increment with argument " << x << endl;
    return x + 1;
}

int main() {
    int x = increment(increment(2));
    cout << "Result:" << x << endl << "Log:" << endl << log.str ();
}
```

Fiecare apel al lui increment produce un mesaj. Mesajele se acumulează.

Stare în C

```
#include <iostream>
using namespace std;
int calls ;
int increment(int x) {
    calls++;
    return x + calls ;
}

int main() {
    int x = increment(increment(2));
    cout << "Result:_" << x << endl << "#Calls:_" << calls << endl;
}
```

Fiecare apel al lui increment citește starea existentă și o modifică.

Logging în Haskell

Funcția originală

```
increment :: Int -> Int  
increment x = x + 1
```

Funcție cu logging

„Îmbogățim” la rezultatul funcției cu mesajul de log.

```
logIncrement :: Int -> (Int, String)  
logIncrement x = (x + 1, "Called increment with argument "  
  ++ show x ++ "\n")
```

Logging în Haskell

Funcția originală

```
increment :: Int -> Int  
increment x = x + 1
```

Funcție cu logging

„Îmbogățim” la rezultatul funcției cu mesajul de log.

```
logIncrement :: Int -> (Int, String)  
logIncrement x = (x + 1, "Called increment with argument "  
  ++ show x ++ "\n")
```

Problemă: Cum calculăm „logIncrement (logIncrement x)”?

Stare în Haskell

Funcția originală în C

```
int increment(int x) {
    return x + calls++;
}
```

Funcția cu stare în Haskell

Rezultatul este acum o funcție, care dată fiind starea dinaintea execuției, produce un rezultat (folosind eventual starea) și starea cea nouă.

```
type State = Integer
stateIncrement :: Int -> (State -> (Int , State))
stateIncrement x = f
    where f calls = (x+calls , calls+1)
```

Problemă: Cum calculăm "stateIncrement (stateIncrement x)"?

Comutații nedeterminate

Exemplu matematic

- Ecuația $y^2 = x$ are două soluții: \sqrt{x} și $-\sqrt{x}$ (complexe, dacă $x < 0$).
- Putem rezolva $z^4 = x$ folosind rezolvarea de mai sus, astfel:
 - Notăm $y = z^2$ și rezolvăm $y^2 = x$ cu soluțiile \sqrt{x} și $-\sqrt{x}$
 - Pentru fiecare soluție s rezolvăm $z^2 = s$, obținând $\sqrt{\sqrt{x}}$, $-\sqrt{\sqrt{x}}$, $\sqrt{-\sqrt{x}}$ și $-\sqrt{-\sqrt{x}}$

Comutație nedeterminată în Haskell

Rezultatul funcției e listă tuturor valorilor posibile.

```
ndSqrt :: Floating a => a -> [a]
ndSqrt x = [sqrt x, -(sqrt x)]
```

Problemă: Cum calculăm "ndSqrt (ndSqrt x)"?

Cum compunem funcții cu efecte laterale

Problema generală

Data fiind funcția $f :: a \rightarrow m\ b$ și funcția $g :: b \rightarrow m\ c$, vreau să obțin o funcție $g \# f :: a \rightarrow m\ c$ care este „compunerea” lui g și f , propagând efectele laterale.

Soluție

Transform $g :: b \rightarrow m\ c$ în $\text{bind } g :: m\ b \rightarrow m\ c$ pentru un `bind` „cu proprietăți bune”

```
bind :: (b -> m c) -> m b -> m c
(#) :: (b -> m c) -> (a -> m b) -> (a -> m c)
g # f = bind g . f
```

Monade

Clasa de tipuri **Monad**

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

- $m\ a$ — tipul **compuțațiilor** care produc rezultate de tip a (și au efecte laterale)
- Tipul $a \rightarrow m\ b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- $(>>=)$ este operația de „secvențiere” a compuțațiilor
- $\text{bind } k\ ma = ma\ >>= k$, adică $\text{bind} = \mathbf{flip}\ (>>=)$
- $(g\ \# f)\ x = (\text{bind } g\ .\ f)\ x = \text{bind } g\ (f\ x) = f\ x\ >>= g$

Logging în Haskell

Funcție cu logging

```
data Log a = Log { val :: a, log :: String }
```

```
logIncrement :: Int -> Log Int
```

```
logIncrement x = Log (x + 1) ("Called increment with  
argument " ++ show x ++ "\n")
```

```
logIncrement2 :: Int -> Log Int
```

```
logIncrement2 x = logIncrement x >=> logIncrement
```

Logging în Haskell

Funcție cu logging

```
data Log a = Log { val :: a, log :: String }
```

```
logIncrement :: Int -> Log Int
```

```
logIncrement x = Log (x + 1) ("Called increment with  
argument " ++ show x ++ "\n")
```

```
logIncrement2 :: Int -> Log Int
```

```
logIncrement2 x = logIncrement x >>= logIncrement
```

Instanța Monad pentru Log

```
instance Monad Log where
```

```
  return a = Log a ""
```

```
  ma >>= k = Log { val = val mb, log = log ma ++ log mb }
```

```
    where mb = k (val ma)
```

Stare în Haskell

Funcția cu stare în Haskell

```
data State state val = State { apply :: state -> (val , state) }
```

```
stateIncrement :: Int -> State Int Int
```

```
stateIncrement x = State f
```

```
  where f calls = (x+calls , calls+1)
```

```
stateIncrement2 :: Int -> State Int Int
```

```
stateIncrement2 x = stateIncrement x >=> stateIncrement
```

Stare în Haskell

Funcția cu stare în Haskell

```
data State state val = State { apply :: state -> (val , state) }
```

```
stateIncrement :: Int -> State Int Int
```

```
stateIncrement x = State f
```

```
  where f calls = (x+calls , calls+1)
```

```
stateIncrement2 :: Int -> State Int Int
```

```
stateIncrement2 x = stateIncrement x >=> stateIncrement
```

Instanța Monad pentru stare

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

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

```
  ma >=> k = State g
```

```
    where g state = let (val , state ') = apply ma state  
                  in apply (k val) state '
```

Comutații nedeterministe

Comutație nedeterministă în Haskell

Rezultatul funcției e listă tuturor valorilor posibile.

```
ndSqrt :: Floating a => a -> [a]  
ndSqrt x = [sqrt x, -(sqrt x)]
```

```
ndSqrt2 :: Floating a => a -> [a]  
ndSqrt2 x = ndSqrt x >=> ndSqrt
```


Comutații nedeterministe

Comutație nedeterministă în Haskell

Rezultatul funcției e listă tuturor valorilor posibile.

```
ndSqrt :: Floating a => a -> [a]  
ndSqrt x = [sqrt x, -(sqrt x)]
```

```
ndSqrt2 :: Floating a => a -> [a]  
ndSqrt2 x = ndSqrt x >=> ndSqrt
```

Instanța Monad pentru liste

```
instance Monad [] where  
  return a = [a]  
  xs >=> k = [y | x <- xs, y <- f x]
```

Proprietățile monadelor

Pe scurt

Operația de compunere $\#$ a continuărilor este asociativă și are element neutru **return**

Pe mai puțin scurt

NeutruD $(\text{return } x) \gg= g = g \ x$

NeutruS $x \gg= \text{return} = x$

Assoc $(f m \gg= g) \gg= h = f m \gg= \lambda x \rightarrow (g \ x \gg= h)$

Să ne definim propria monadă IO

Monada MyIO

partea I

```
module MyIO(MyIO, myPutChar, myGetChar, convert) where

type Input = String
type Output = String

data MyIO a = MyIO { apply :: Input -> (a, Input, Output) }
```

Observație: Tipul MyIO is abstract

- Sunt exportate doar tipul MyIO, myPutChar, myGetChar, convert (și operațiile de monadă)
- Nu este exportat constructorul MyIO și nici operația apply

Monada MyIO

partea II

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

myGetChar :: MyIO Char
myGetChar = MyIO (\ (ch:input') -> (ch, input', ""))
```

Exemplu

```
apply myGetChar "abc" == ('a', "bc", "")
apply myGetChar "bc"  == ('b', "c", "")
apply (myPutChar 'A') "def" == ((), "def", "A")
apply (myPutChar 'B') "def" == ((), "def", "B")
```

Monada MyIO

partea III

```
instance Monad MyIO where
  return x = MyIO (\ input -> (x, input, ""))
  m >>= k   = MyIO (\ input ->
    let (x, input1, output1) = apply m input
        (y, input2, output2) = apply (k x) input1
    in (y, input2, output1 ++ output2))
```

Exemplu

```
apply
  (myGetChar >>= \x -> myGetChar >>= \y -> return [x,y])
  "abc"
== ("ab", "c", "")
apply (myPutChar 'A' >> myPutChar 'B') "def"
== ((), "def", "AB")
apply (myGetChar >>= \x -> myPutChar (toUpper x)) "abc"
== ((), "bc", "A")
```

Monada MyIO

partea IV

```
convert :: MyIO () -> IO ()
convert m = interact (\ input ->
                      let (x, input', output) = apply m input
                      in output)
```

Unde

```
interact :: (String -> String) -> IO ()
```

face parte din biblioteca standard, si 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)

Folosirea monadei MyIO

partea I

```
module MyEcho where
```

```
import Char
```

```
import MyIO
```

```
myPutStr :: String -> MyIO ()
```

```
myPutStr = foldr (>>) (return ()) . map myPutChar
```

```
myPutStrLn :: String -> MyIO ()
```

```
myPutStrLn s = myPutStr s >> myPutChar '\n'
```


Folosirea monadei MyIO

partea II

```

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

myEcho :: MyIO ()
myEcho = myGetLine >>= \line ->
    if line == "" then
        return ()
    else
        myPutStrLn (map toUpper line) >>
            myEcho

main :: IO ()
main = convert myEcho

```

În execuție

partea I

```
10-monade$ runghc MyEcho
```

```
This is a test.
```

```
THIS IS A TEST.
```

```
It is only a test.
```

```
IT IS ONLY A TEST.
```

```
Were this a real emergency, you'd be dead now.
```

```
WERE THIS A REAL EMERGENCY, YOU'D BE DEAD NOW.
```

```
10-monade$
```

Folosind notația **do**

```

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

myEcho :: MyIO ()
myEcho = do {
    line <- myGetLine;
    if line == "" then
        return ()
    else do {
        myPutStrLn (map toUpper line);
        myEcho
    }
}

```

Descrieri de liste și monade

Monada listelor

Definiția în biblioteca standard:

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

```
instance Monad [] where
```

```
  return          :: a -> [a]
  return x        = [ x ]

  (>>=)           :: [a] -> (a -> [b]) -> [b]
  m >>= k         = [ y | x <- m, y <- k x ]
```

Recursiv:

```
[ ] >>= k          = [ ]
(x:xs) >>= k       = (k x) ++ (xs >>= k)
```

Cu funcții de ordin înalt:

```
m >>= k = concat (map k m)
```

Descrieri de liste și monada listelor

Notăție **do**

```

pairs :: Int -> [(Int, Int)]
pairs n = [ (i, j) | i <- [1..n], j <- [(i+1)..n] ]

```

este echivalentă cu

```

pairs' :: Int -> [(Int, Int)]
pairs' n = do {
    i <- [1..n];
    j <- [(i+1)..n];
    return (i, j)
}

```

Exemplu:

```

*Main> pairs 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
*Main> pairs' 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]

```

Monade cu structură de monoid

Definiția în biblioteca standard:

```

class Monad m => MonadPlus m where
    mzero  :: m a
    mplus  :: m a -> m a -> m a

instance MonadPlus [] where
    mzero    :: [a]
    mzero    = []

    mplus    :: [a] -> [a] -> [a]
    mplus    = (++)

guard  :: MonadPlus m => Bool -> m ()
guard False = mzero
guard True  = return ()

msum  :: MonadPlus m => [m a] -> m a
msum = foldr mplus mzero
  
```

Descrieri de liste cu filtrare

```

pairs '' :: Int -> [(Int, Int)]
pairs '' n = [ (i,j) | i <- [1..n], j <- [1..n], i < j ]

```

este echivalentă cu

```

pairs ''' :: Int -> [(Int, Int)]
pairs ''' n = do {
    i <- [1..n];
    j <- [1..n];
    guard (i < j);
    return (i,j)
}

```

Exemplu

```

*Main> pairs '' 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
*Main> pairs ''' 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]

```


Analiză sintactică

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

A doua încercare

```
type Parser a = String -> (a, String)
```

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

A doua încercare

```
type Parser a = String -> (a, String)
```

- Dar dacă gramatica e ambiguă?
- Dar dacă intrarea nu corespunde nici unui element din a?

Tipul unui analizor sintactic

A treia încercare

Dr. Seuss on Parser Monads:



```
type Parser a - String → [(a,String)]
```

A Parser for Things
is a function from Strings
to Lists of Pairs
of Things and Strings!

Art: Seuss; Type: Wagner, Rhyme: Roehr

Modulul Parser

partea I

```
module Parser (Parser , apply , parse , char , spot , token ,
    star , plus , parseInt) where
```

```
import Char
import Monad
```

```
-- Tipul (incapsulat) Parser
```

```
newtype Parser a = Parser (String -> [(a, String)])
```

```
-- Folosirea unui parser (functie privata)
```

```
apply :: Parser a -> String -> [(a, String)]
```

```
apply (Parser f) s = f s
```

```
-- Daca exista parsare, da prima varianta
```

```
parse :: Parser a -> String -> a
```

```
parse m s = head [ x | (x,t) <- apply m s, t == "" ]
```

Modulul Parser

partea II: Parser e monadă

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

```
instance Monad Parser where
  return x = Parser (\s -> [(x,s)])
  m >>= k = Parser (\s ->
    [ (y, u) |
      (x, t) <- apply m s,
      (y, u) <- apply (k x) t ])
```


Modulul Parser

partea II: Parser e monadă cu plus

```
-- class MonadPlus m where  
--   mzero  :: m a  
--   mplus  :: m a -> m a -> m a
```

```
instance MonadPlus Parser where  
  mzero      = Parser (\s -> [])  
  mplus m n  = Parser (\s -> apply m s ++ apply n s)
```

- mzero reprezintă analizorul sintactic care eșuează tot timpul
- mplus reprezintă combinarea alternativelor

Parsare pentru caractere

-- *Recunoasterea unui caracter*

char :: Parser **Char**

char = Parser f

where

f [] = []

f (c:s) = [(c,s)]

-- *Recunoasterea unui caracter cu o proprietate*

spot :: (**Char** -> **Bool**) -> Parser **Char**

spot p = Parser f

where

f [] = []

f (c:s) | p c = [(c, s)]
| **otherwise** = []

-- *Recunoasterea unui anumit caracter*

token :: **Char** -> Parser **Char**

token c = spot (== c)

Recunoașterea unui caracter cu o proprietate

Gărzi și notăție **do**

```
spot :: (Char -> Bool) -> Parser Char
spot p = Parser f
  where
    f [] = []
    f (c:s) | p c = [(c, s)]
              | otherwise = []
```

e echivalentă cu

```
spot :: (Char -> Bool) -> Parser Char
spot p = do { c <- char; guard (p c); return c }
```

Recunoașterea unui cuvânt cheie

```
match :: String -> Parser String
match []      = return []
match (x:xs) = do
    y <- token x;
    ys <- match xs;
    return (y:ys)
```

Recunoașterea unei secvențe repetitive

```

-- Steluta Kleene (zero, una sau mai multe repetitii)
star :: Parser a -> Parser [a]
star p = plus p 'mplus' return []

-- cel puțin o repetitie
plus :: Parser a -> Parser [a]
plus p = do { x <- p;
              xs <- star p;
              return (x:xs) }

```

Recunoașterea unui număr întreg

-- *Recunoașterea unui număr natural*

parseNat :: Parser Int

```
parseNat = do { s <- plus (spot isDigit);
               return (read s) }
```

-- *Recunoașterea unui număr negativ*

parseNeg :: Parser Int

```
parseNeg = do { token '-';
               n <- parseNat
               return (-n) }
```

-- *Recunoașterea unui număr întreg*

parseInt :: Parser Int

```
parseInt = parseNat 'mplus' parseNeg
```

Modulul Exp

```
module Exp where
```

```
import Monad
```

```
import Parser
```

```
data Exp = Lit Int  
    | Exp :+: Exp  
    | Exp :*: Exp  
    deriving (Eq, Show)
```

```
evalExp    :: Exp -> Int
```

```
evalExp    (Lit n)      = n
```

```
evalExp    (e :+: f)    = evalExp e + evalExp f
```

```
evalExp    (e :*: f)    = evalExp e * evalExp f
```

Recunoașterea unei expresii

```

parseExp :: Parser Exp
parseExp = parseLit 'mplus' parseAdd 'mplus' parseMul
  where
    parseLit = do { n <- parseInt;
                    return (Lit n) }
    parseAdd = do { token '(';
                    d <- parseExp;
                    token '+';
                    e <- parseExp;
                    token ')';
                    return (d :+: e) }
    parseMul = do { token '(';
                    d <- parseExp;
                    token '*';
                    e <- parseExp;
                    token ')';
                    return (d :*: e) }

```


Recunoașterea unei expresii

Test

```
*Exp> parse parseExp "(1+(2*3))"
Lit 1 :+: (Lit 2 *: Lit 3)
*Exp> evalExp (parse parseExp "(1+(2*3))")
7
*Exp> parse parseExp "((1+2)*3)"
(Lit 1 :+: Lit 2) *: Lit 3
*Exp> evalExp (parse parseExp "((1+2)*3)")
9
```