Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ana Cristina Țurlea

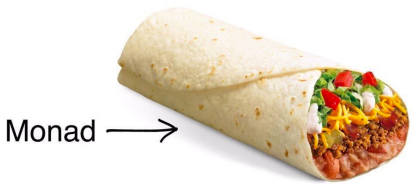
ana.turlea@fmi.unibuc.ro

Ce este o monadă?

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este un burrito. https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/



https://twitter.com/monadburritos

- "All told, a monad in X is just a monoid in the category of endofunctors in X, with product x replaced by composition of endofunctors and unit set by the identity endofunctor."
  Saunders Mac Lane, Categories for the Working Mathematician, 1998.

Funcții "îmbogățite" și programarea cu efecte

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$

  știind x, obținem direct y

- Funcție îmbogățită: $x \mapsto$ 

  știind x, putem să extragem y și producem un efect

Referințe:

https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/

https://bartoszmilewski.com/2016/11/30/monads-and-effects/

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$ 

Exemple

- Folosind tipul **Maybe** a

```
data Maybe a = Nothing | Just a
f :: Int -> Maybe Int
f  x = if x < 0 then Nothing else (Just x)
```

- Folosind un tip care are ca efect un mesaj

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

f :: Int -> Writer String Int
f  x = if x < 0 then (Writer (-x, "negativ"))
                else (Writer (x, "pozitiv"))
```

Logging în Haskell

„Îmbogățim” rezultatul funcției cu mesajul de log.

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

Observații

- datele de tip Writer **log** a sunt definite folosind înregistrări
- o dată de tip Writer **log** a are una din formele
  Writer (va,vlog) sau Writer {runWriter = (va,vlog)}
  unde va :: a și vlog :: **log**
- runWriter este funcția proiecție:
  runWriter :: Writer **log** a –> (a, **log**)
  de exemplu runWriter (Writer (1,"msg")) = (1,"msg")

Compunerea funcțiilor

- Principala operație pe care o facem cu funcții este compunerea

```
f :: a -> b , g :: b -> c , g . f :: a -> c
(.) :: (b -> c) -> (a -> b) -> a -> c
```

  Valoarea de tip b este transmisă direct funcției g.

- Ce facem dacă

```
f :: a -> m b , g :: b -> m c
```

  unde m este un constructor de tip care îmbogățește tipul?

  De exemplu,

  - m = **Maybe**
  - m = Writer **log**
    Atenție! m trebuie să aibă un singur argument.

Compunerea funcțiilor

```
f :: a -> m b , g :: b -> m c
```

unde m este un constructor de tip care îmbogățește tipul.

Vrem să definim o "compunere" pentru funcții îmbogățite

```
(<=<) :: (b -> m c) -> (a -> m b) -> a -> m c
```

Atunci când definim g <=< f trebuie să extragem valoarea întoarsă de f și să o trimitem lui g.

Exemplu: logging în Haskell

„Îmbogățim” rezultatul funcției cu mesajul de log.

```
newtype Writer log a = Writer { runWriter :: (a, log) }
-- newtype seamana cu data, se foloseste cand avem un singur
-- constructor  si un singur tip de date

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

Problemă: Cum calculăm logIncrement (logIncrement x)?

```
logIncrement2 :: Int -> Writer String Int
logIncrement2 x = let (y, log1) = runWriter (logIncrement x)
                      (z, log2) = runWriter (logIncrement y)
                  in  Writer (z, log1 ++ log2)
```

Se poate! ... dar nu vrem să facem asta pentru fiecare funcție.

Cum compunem funcții cu efecte laterale

Problema generală

Dată fiind funcția f :: a -> m b și funcția g :: b -> m c, vreau să obțin o funcție g <=< f :: a -> m c care este „compunerea” lui g și f, propagând efectele laterale.

Exemplu

```
> logIncrement x = Writer (x + 1, "Called increment with
    argument " ++ show x ++ "\n")
> logIncrement <=< logIncrement $ 3
Writer {runWriter = (5,"Called increment with argument 3\
    nCalled increment with argument 4\n")}
```

Observație Funcția (<=<) este definită în Control.**Monad**

Monade în Haskell

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
- în Control.**Monad** sunt definite
  - f >**=>**g = \x -> f x >>= g
  - (<=<) = **flip** (>**=>**)

Applicative va fi discutată mai târziu

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

În Haskell, monada este o clasă de tipuri!

Exemple: monade predefinite

- monada **Maybe**

```
> (lookup 3 [(1,2), (3,4)]) >>= (\x -> if (x<0) then
    Nothing else (Just x))
Just 4


> (lookup 3 [(1,2), (3,-4)]) >>= (\x -> if (x<0) then
    Nothing else (Just x))
Nothing


> (lookup 3 [(1,2)]) >>= (\x -> if (x<0) then Nothing
    else (Just x))
Nothing
```

Exemple: monade predefinite

- monada listelor

```
> f = (\x -> if (x>=0) then [sqrt x,-sqrt x)] else [])

> [4,8] >>= f
[2.0,-2.0,2.8284271247461903,-2.8284271247461903]

> [4,8] >>= f >>= f
[1.4142135623730951,-1.4142135623730951,
1.6817928305074292,-1.6817928305074292]
```

Proprietățile monadelor

Asociativitate și element neutru

Operația <=< de compunere a funcțiilor îmbogățite este asociativă și are element neutru **return**

- Element neutru (la dreapta): g <=< **return** = g

```
(return x) >>= g = g x
```

- Element neutru (la stânga): **return** <=< g = g

```
x >>= return = x
```

- Associativitate: h <=< (g <=< f) = (h <=< g) <=< f

```
(f >>= g) >>= h = f >>= (\ x -> (g x >>= h))
```

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
```

Notația **do** pentru monade

De exemplu

```
e1    >>= \x1 ->
e2    >>= \x2 ->
e3    >>= \_  ->
e4    >>= \x4 ->
e5
```

devine

```
do
    x1 <- e1
    x2 <- e2
    e3
    x4 <- e4
    e5
```

Functor / Applicative / **Monad**

Functor: efecte laterale

Functor

Schimbă rezultatul: efectele laterale rămân aceleași

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Exemplu — liste

Dată fiind o funcție `f :: a -> b` și o listă *la* de elemente de tip a, vreau să să obțin o lista de elemente de tip b transformând fiecare element din *la* folosind funcția `f`.

```
instance Functor [] where
  fmap = map
```

Functor: efecte laterale

Functor

Schimbă rezultatul: efectele laterale rămân aceleași

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Instanță pentru tipul optiune

fmap :: (a –> b) –> **Maybe** a –> **Maybe** b

```
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

Clasa de tipuri Functor

Instanțe

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Instanță pentru tipul eroare
fmap :: (a -> b) -> **Either** e a -> **Either** e b

```
instance Functor (Either e) where
    fmap _ (Left x) = Left x
    fmap f (Right y) = Right (f y)
```

Instanță pentru tipul funcție
fmap :: (a -> b) -> (t -> a) -> (t -> b)

```
instance Functor (->) a where
  fmap f g = f . g  -- sau, mai simplu, fmap = (.)
```

Exemple

```
Main> fmap (*2) [1..3]
[2,4,6]
Main> fmap (*2) (Just 200)
Just 400
Main> fmap (*2) Nothing
Nothing
Main> fmap (*2) (+100) 4
208
Main> fmap (*2) (Right 6)
Right 12
Main> fmap (*2) (Left 135)
Left 135
Main> fmap (show . (*2) . read) getLine >>= putStrLn
123
246
```

Problemă

- Folosind fmap putem transforma o funcție `h :: a -> b` într-o funcție între computații cu efecte `fmap h :: m a -> m b`
- Dar ce se întâmplă dacă avem o funcție cu mai multe argumente E.g., cum trecem de la `h :: a -> b -> c` la `h' :: m a -> m b -> m c`?
- Putem încerca să folosim fmap
- dar, deoarece `h :: a -> (b -> c)` obtinem
  `fmap h :: m a -> m (b -> c)`
- Putem aplica `fmap h` la o valoare `ca :: m a` și obținem
  `fmap h ca :: m (b -> c)`

Problemă

Cum transformăm un obiect din `m (b -> c)` într-o funcție `m b -> m c`?

Clasa de tipuri Applicative

Definiție

```
class Functor m => Applicative m where
  pure :: a -> m a
  (<*>) :: m (a -> b) -> m a -> m b
```

- Orice instanță a lui Applicative trebuie să fie instanță a lui **Functor**

Instanță pentru tipul opțiune

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  Just f  <*> x = fmap f x
```

Clasa de tipuri Applicative

Instanță pentru tipul opțiune

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  Just f  <*> x = fmap f x
```

```
> pure "Hey" :: Maybe String
Just "Hey"
> (++) <$> (Just "Hey ") <*> (Just "You!")
Just "Hey You!"
```

<$> este operatorul infix echivalent cu `fmap`

Tipul listă (computație nedeterministă)

Instanță pentru tipul computațiilor nedeterministe (liste)

```
instance Applicative [] where
  pure x = [x]
  fs  <*> xs = [f x | f <- fs, x <- xs]
```

```
Main> pure "Hey" :: [String]
["Hey"]
Main> (++) <$> ["Hello ", "Goodbye "] <*> ["world", "
    happiness"]
["Hello world","Hello happiness","Goodbye world","Goodbye
    happiness"]
Main> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
Main> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

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

Moanda IO

Monada IO

```
data IO a = IO a -- reprezinta atat o valoare de tip a
                 -- cat si un efect

data () = () -- tipul unitate

getChar :: IO Char

putChar :: Char -> IO ()

main :: IO ()
main = do c <- getChar
          putChar c

return :: a -> IO a

getLine :: IO String
getLine = do c <- getChar
             if c == '\n' then return ""
             else do l <- getLine
                     return (c:l)
```

Monada IO

Putem combina două comenzi

```
(>>) :: IO () -> IO () -> IO ()
```

```
putChar '?' >> putChar '!'
?!
```

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

```
getChar >>= \x -> putChar (toUpper x)
c
C
```

Monada IO

```
putStrRec :: String -> IO ()
putStrRec [] = return ()
putStrRec (x:xs) = putChar x >> putStrRec xs
```

```
putStrHOF :: String -> IO ()
putStrHOF = foldr (>>) (return ()) . map putChar
```

```
myGetLine :: IO String
myGetLine = getChar >>= \x ->
                    if x == '\n' then
                      return []
                    else
                      myGetLine >>= \xs -> return (x:xs)
```

Monada IO

```
myGetLine :: IO String
myGetLine = getChar >>= \x ->
                        if x == '\n' then
                          return []
                        else
                          myGetLine >>= \xs -> return (x:xs)
```

```
getLineDO :: IO String
getLineDO = do
       x <- getChar
       if x == '\n' then
         return []
       else do
              xs <- getLineDO
              return (x:xs)
```

Monada IO

```
showInteger :: Int -> IO ()
showInteger n = putStr (show n )

showPlusZece n = showInteger n >> return (n + 10)
```

Monada IO: Read

Dacă am vrea să citim un întreg de la tastatură pentru a îl trimite unei funcții, ar trebui întâi să putem transforma un șir de caractere într-un număr întreg. Pentru asta folosim funcția **read**:

```
read :: Read a => String -> a
```

```
Prelude> read "10" + 10
20
Prelude> read "False" || True
True
Prelude> read "[1,2,3]" ++ [4]
[1,2,3,4]
Prelude> read "1" || True
*** Exception: Prelude.read: no parse
```

Compilatorul va încerca să deducă tipul pe care trebuie să îl întoarcă funcția **read** din contextul în care este folosit. `Atentie`: **read** va arunca o excepție dacă șirul primit nu poate fi convertit la tipul dedus din context.

getLine și read

Vrem să citim un întreg de la tastatură, apoi să aplicăm funcția showPlusZece:

```
citesteSiAdunaZece :: IO Integer
citesteSiAdunaZece = ?? read getLine >>= showPlusZece

read :: Read a => String -> a
getLine :: IO String
-- Ce tip are ??
-- Cu ce functie seamana ??
??   :: (String -> Integer) -> IO String -> IO Integer
map  :: (a      -> b      ) -> [a]        -> [b]
fmap :: (a      -> b      ) -> f a        -> f b


citesteSiAdunaZece :: IO Integer
citesteSiAdunaZece = fmap read getLine >>= showPlusZece
```

Adunarea în IO

Vrem să citim două numere de la tastatură și să întoarcem suma lor.

```
suma :: IO Integer
suma =
    fmap read getLine >>=
    (\x -> fmap read getLine >>=
        (\y -> ?? (x + y)
        )
    )


??     ::                Integer -> IO Integer
return :: Monad m => a           -> m  a
```

Adunarea în IO

Vrem să citim două numere de la tastatură și să întoarcem suma lor.

```
suma :: IO Integer
suma =
    fmap read getLine >>=
    (\x -> fmap read getLine >>=
        (\y -> return (x + y)
        )
    )
```

```
??     ::                Integer -> IO Integer
return :: Monad m => a           -> m  a
```

Putem folosi **return** pentru a introduce valori pure în **IO**. Deși putem scrie funcții folosind >>=, codul devine greu de citit.

Notația **do**

```
suma :: IO Integer
suma =
    fmap read getLine >>= (\x ->
    fmap read getLine >>= (\y ->
    return (x + y)
    ))
```

Funcția suma poate fi scrisă echivalent folosind notația **do**:

```
suma :: IO Integer
suma = do
    x <- fmap read getLine
    y <- fmap read getLine
    return (x + y)
```

Citire și scriere

Notația **do** ne permite de asemenea să combinăm scrierea și citirea:

```
citesteSiScrie :: IO Integer
citesteSiScrie = do
    putStrLn "Introduceti numele: "
    nume <- getLine
    putStrLn ("Buna ziua, " ++ nume)
    putStrLn "Introduceti doua numere:"
    x <- fmap read getLine
    y <- fmap read getLine
    let suma = x + y
    putStrLn ("Suma este: " ++ show suma)
    return suma
```

putStrLn este identic cu **putStr**, cu excepția că adaugă un sfârșit de linie.

Citire și scriere cu fișiere

În Haskell, lucrul cu fișierele este foarte similar lucrului cu consola:

```
type FilePath = String

readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Spre exemplu, putem scrie o funcție care citește un fișier și îl printează în consolă:

```
cat :: FilePath -> IO ()
cat path = do
  continut <- readFile path
  putStr continut
```

Exemplu

Exemplu: citirea unui fișier de intrare și convertirea tuturor caracterelor in majuscule:

```
toUpperFile :: IO ()
toUpperFile = do
  putStr "Fisier intrare: "
  inPath <- getLine

  continut <- readFile inPath
  let caps = map toUpper continut

  putStr "Fisier iesire: "
  outPath <- getLine

  writeFile outPath caps
```

Suma numerelor dintr-un fișier

```
lines :: String -> [String]
words :: String -> [String]
concat :: [[String]] -> [String]

sumaFisier :: FilePath -> IO Integer
sumaFisier path = do
    continut <- readFile path
    let linii = lines continut      -- linii   :: [String]
    let cuvinte = map words linii   -- cuvinte :: [[String]]
    let strNum = concat cuvinte     -- strNum  :: [String]
    let numere = map read strNum    -- numere  :: [Integer]
    return (sum numere)
```

Programul va calcula suma numerelor din fișierul de intrare. Acestea pot fi separate de spații sau linii.

Pe săptămâna viitoare!