

Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Ana Cristina Turlea

Departamentul de Informatică, FMI, UB
ioana@fmi.unibuc.ro
ana.turlea@fmi.unibuc.ro

- 1 Ce este o monadă?
- 2 Funcții "îmbogățite" și programarea cu efecte
- 3 Monade în Haskell
- 4 **Functor** / **Applicative** / **Monad**

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**

$$f :: a \rightarrow b, g :: b \rightarrow c, g . f :: a \rightarrow c$$

$$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$

Valoarea de tip b este transmisă **direct** funcției g .

- Ce facem dacă

$$f :: a \rightarrow m\ b, g :: b \rightarrow m\ c$$

unde m este un **constructor de tip** care îmbogățește tipul?

De exemplu,

- $m = \text{Maybe}$
- $m = \text{Writer } \text{log}$

Atenție! m trebuie să aibă un singur argument.

Compunerea funcțiilor

$f :: a \rightarrow m\ b$, $g :: b \rightarrow m\ c$

unde m este un **constructor de tip** care îmbogățește tipul.

Vrem să definim o "compunere" pentru funcții îmbogățite

$$(<=<) :: (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ b) \rightarrow a \rightarrow m\ c$$

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

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.

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)
- $a \rightarrow m\ b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- $>>=$ este operația de „secvențiere” a computațiilor
- în `Control.Monad` sunt definite
 - $f\ >=>g = \lambda x \rightarrow f\ x\ >>= g$
 - $(<=<) = \text{flip } (>=>)$

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)
- $a \rightarrow m\ b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- $>=$ este operația de „secvențiere” a computațiilor

În Haskell, monada este o clasă de tipuri!

Kinds (tipuri de tipuri)

Observăm că `m` în definiția de mai sus este un **constructor de tip**.

În Haskell, valorile sunt clasificate cu ajutorul *tipurilor*:

```
Prelude> :t "as"  
"as"  :: [Char]
```

Constructorii de tipuri sunt la rândul lor clasificați în *kind-uri*:

```
Prelude> :k Char  
*      -- constructor de tip fara argumente  
Prelude> :k []  
[] :: * -> *      -- constructor de tip cu un argument
```

Constructorii de tip pot fi și ei grupați în clase.

Example: 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
```

Example: 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): $g \leq\leq \text{return} = g$

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

- Element neutru (la stânga): $\text{return} \leq\leq g = g$

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

- Asociativitate: $h \leq\leq (g \leq\leq f) = (h \leq\leq g) \leq\leq f$

$$(f \gg= g) \gg= h = f \gg= (\backslash x \rightarrow (g \ x \gg= h))$$

Notăția **do** pentru monade

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

De exemplu

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

devine

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

Notăția **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

Data fiind o funcție $f :: a \rightarrow b$ și o listă la de elemente de tip a , vreau 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 = (.)
```

Example

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

Problemă

Cum transformăm un obiect din $m\ (b \rightarrow c)$ într-o funcție $m\ b \rightarrow m\ c$?

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 nedeterminate (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
```

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

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

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

Pe săptămâna viitoare!