Programare funcțională

Introducere în programarea funcțională folosind Haskell
C09

Claudia Chiriță
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Mini Workshop de PF

Evie Ciobanu

Senior Software Engineer

Prisma

Ana Pantilie

Haskell Engineer

IOHK

Julian Sutherland

Head of Formal Verification

Nethermind

Monade - privire de ansamblu

Despre intenție și acțiune

[1] S. Peyton-Jones, Tackling the Awkward Squad: ...

- [1] A purely functional program implements a function; it has no side effect.
- [1] Yet the ultimate purpose of running a program is invariably to cause some side effect: a changed file, some new pixels on the screen, a message sent, ...

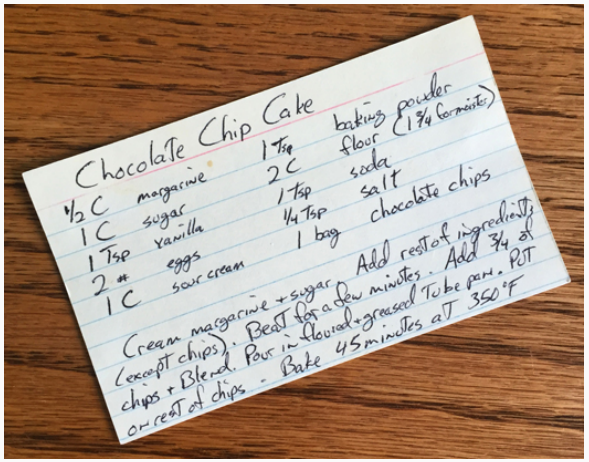
Exemplu

```
putChar :: Char -> IO ()
Prelude> putChar '!'
```

reprezintă o comandă care, dacă va fi executată, va afișa un semn de exclamare.

Mind-Body Problem - Rețetă vs Prăjitură

c :: Cake



r :: Recipe Cake

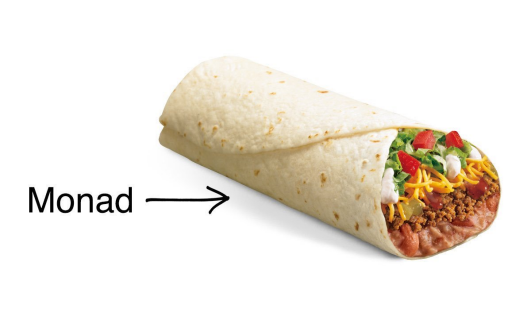
IO este o rețetă care produce o valoare de tip **a**.

Motorul care citește și execută instrucțiunile IO se numește Haskell Runtime System (RTS). Acest sistem reprezintă legatura dintre programul scris și mediul în care va fi executat, împreună cu toate efectele și particularitățile acestuia.

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este o clasă de tipuri în Haskell.
- "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.
- O monadă este un burrito.
  https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/

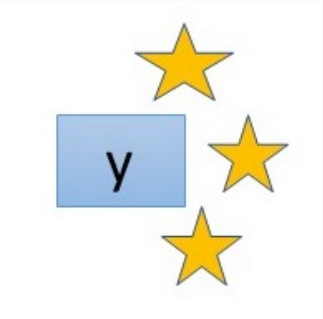


https://twitter.com/monadburritos

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$

  știind x, obținem direct y

- Funcție îmbogățită: $x \mapsto$ 

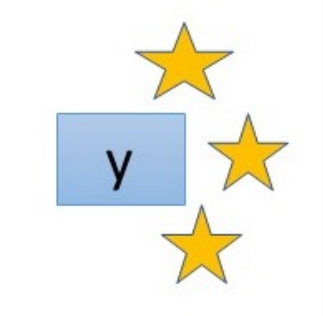
  știind x, putem să extragem y și să producem un efect

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: $x \mapsto$ 

Exemplu:

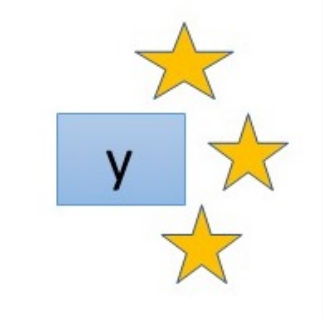
Folosind tipul **Maybe** a

```
data Maybe a = Nothing | Just a
f :: Int -> Maybe Int
f x = if x < 0 then Nothing else (Just x+1)
```

Cum putem calcula f . f?

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$



Exemplu:

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"))
```

Cum putem calcula f.f și să concatenăm mesajele?

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
```

În Haskell, monada este o clasă de tipuri!

Clasa **Monad** este o extensie a clasei Applicative!

- m a este tipul comenzilor 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 comenzilor

Proprietățile monadelor

Asociativitate și element neutru

Operația >>= este asociativă și are element neutru **return**

- Element neutru (la dreapta):

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

- Element neutru (la stânga):

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

- Associativitate:

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

De ce nu este suficient fmap?

Exemplu: **putStrLn <$> getLine**

```
<$> :: Functor f => (a -> b) -> f a -> f b
getLine   :: IO String
putStrLn  :: String -> IO ()
-- in exemplul nostru, b devine IO ()

--                                  [1]  [2]  [3]
putStrLn <$> getLine  :: IO (IO  ()))
```

[1] **IO** din exterior reprezintă efectul pe care **getLine** trebuie să îl producă pentru a obține un **String** introdus de utilizator

[2] **IO** din interior reprezintă efectul care s-ar produce dacă **putStrLn** a fost evaluat

[3] () este tipul unitate întors de **putStrLn**

De ce nu este suficient fmap?

```
putStrLn <$> getLine :: IO (IO ())
```

Trebuie să unim efectele lui **getLine** și **putStrLn** într-un singur efect **IO**!

```
import Control.Monad (join)
join :: Monad m => m (m a) -> m a
join $ putStrLn <$> getLine
```

Notația do pentru monade

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

Notația cu operatori	Notația **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

```
binding' :: IO ()
binding' =
  getLine >>= putStrLn

binding :: IO ()
binding = do
  name <- getLine
  putStrLn name
```

Notația do pentru monade

```
twoBinds' :: IO ()
twoBinds' =
    putStrLn "name pls:" >>
    getLine >>=

    \name ->
     putStrLn "age pls:" >>
     getLine >>=

    \age ->
     putStrLn ("y helo thar: "
             ++ name ++ " who is: "
             ++ age ++ " years old.")
```

Notația do pentru monade

```
twoBinds :: IO ()
twoBinds = do
    putStrLn "name pls:"
    name <- getLine

    putStrLn "age pls:"
    age <- getLine

    putStrLn ("y helo thar: "
              ++ name ++ " who is: "
              ++ age ++ " years old.")
```

Notația do pentru monade

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

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

Notația do pentru monade

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

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

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

De exemplu

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

devine

Notația do pentru monade

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

De exemplu

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

devine

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

Functor și Applicative definiți cu return și >>=

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

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)

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

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
return :: a -> Maybe a

instance Monad Maybe where
  return = Just

  Just va >>= f  = f va
  Nothing >>= _  = Nothing
```

Monada Maybe – exemplu

```haskell
radical :: Float -> Maybe Float
radical x
    | x >= 0 = return (sqrt x)
    | x < 0  = Nothing

-- a * x^2 + b * x + c = 0
solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 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 Maybe – exemplu

Atenție! Acest exemplu folosește și monada listă care o să fie explicat în cursul următor.

```
-- a * x^2 + b * x + c = 0
solEq2All :: Float -> Float -> Float -> Maybe [Float]
solEq2All 0 0 0 = return [0]
solEq2All 0 0 c = Nothing
solEq2All 0 b c = return [negate c / b]
solEq2All a b c = do
        rDelta <- radical (b * b - 4 * a * c)
        let s1 = (negate b + rDelta) / (2 * a)
        let s2 = (negate b - rDelta) / (2 * a)
        return [s1,s2]
```

Pe data viitoare!