Programare funcțională

Introducere în programarea funcțională folosind Haskell
C12

Ana Iova
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

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.

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, ...

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, ...

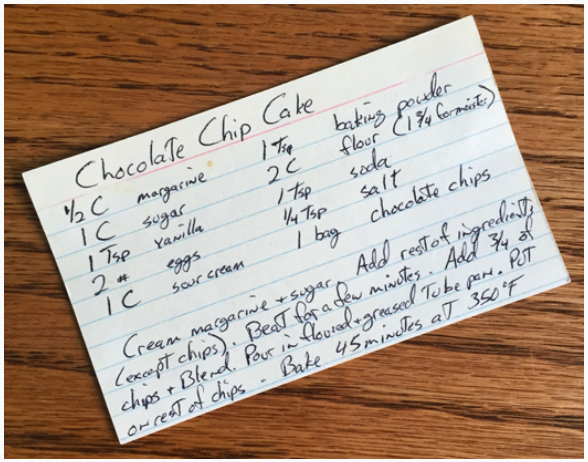
Exemplu

```
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

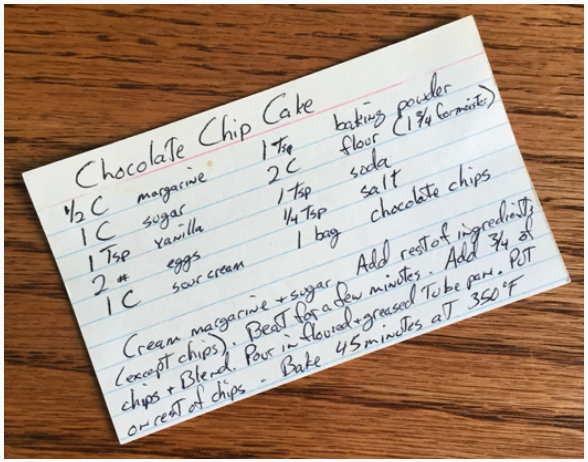
c :: Cake



r :: Recipe Cake

Mind-Body Problem - Rețetă vs Prăjitură

c :: Cake

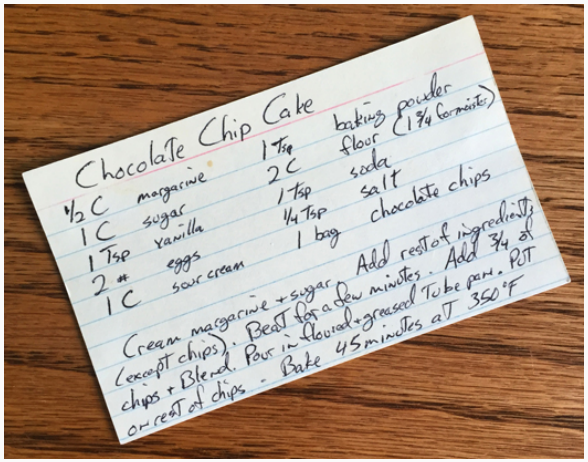


r :: Recipe Cake

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

Mind-Body Problem - Rețetă vs Prăjitură

c :: Cake



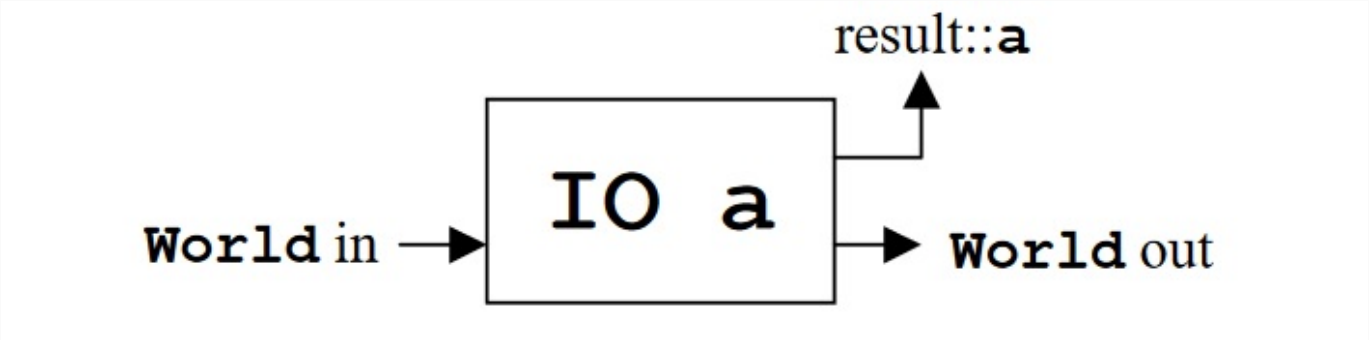
r :: Recipe Cake

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

Motorul care citeste si executa instructiunile IO se numeste Haskell Runtime System (RTS). Acest sistem reprezinta legatura dintre programul scris si mediul în care va fi executat, împreuna cu toate efectele si particularitatile acestuia.

Comenzi în Haskell

```
type IO a = RealWorld -> (a, RealWorld)
```



S. Peyton-Jones, Tackling the Awkward Squad: ...

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este o clasă de tipuri în Haskell.

Ce este o monadă?

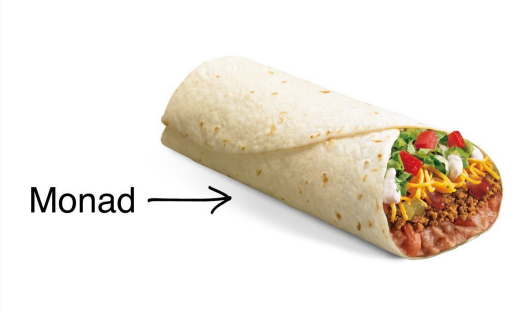
Există multe răspunsuri, variind între

- 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.

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

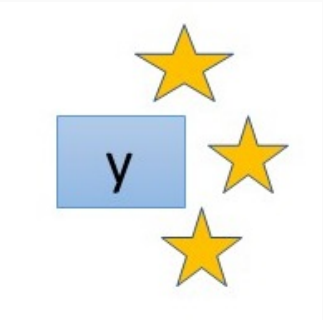
- Funcție simplă: $x \mapsto y$

  știind x, obținem direct y

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$

  știind x, obținem direct y

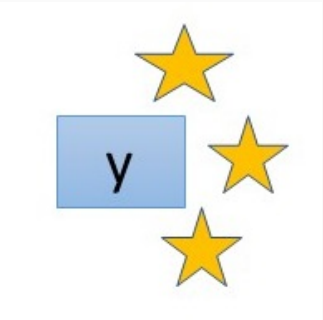
- Funcție îmbogățită: $x \mapsto$ 

  știind x, putem să extragem y și producem un efect

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/

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

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)

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

data Maybe a = Nothing | Just a
```

Monada Maybe(a funcțiilor parțiale)

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

data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Just va >>= k   = k va
  Nothing >>= _   = Nothing
```

Monada listelor (a funcțiilor nedeterministe)

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

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

Rezultatul funcției e lista tuturor valorilor posibile.

Monada IO

- **IO** () corespunde comenzilor care nu produc rezultate
  - () este tipul unitate care conține doar valoarea ()
- **IO** a corespunde comenzilor care produc rezultate de tip a.

Monada IO

- **IO** () corespunde comenzilor care nu produc rezultate
  - () este tipul unitate care conține doar valoarea ()
- **IO** a corespunde comenzilor care produc rezultate de tip a.

Exemplu: citește un caracter

IO Char corespunde comenzilor care produc rezultate de tip **Char**

```
getChar :: IO Char
```

Dacă „șirul de intrare” conține "abc" atunci **getChar** produce

- 'a'
- șirul rămas de intrare "bc"

Combinarea comenzilor cu valori

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

Exemplu

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

- Dacă „șirul de intrare” conține "abc"
- atunci comanda de mai sus, atunci când se execută, produce:
  - ieșirea "A"
  - șirul rămas de intrare "bc"

Operatorul de legare / bind

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

- Dacă fiind o comandă care produce o valoare de tip a
  m :: **IO** a
- Data fiind o funcție care pentru o valoare de tip a se evaluează la o comandă de tip b
  k :: a -> **IO** b
- Atunci
  m >>= k :: **IO** b
  este comanda care, dacă se va executa:
  - Mai întâi efectuează m, obținând valoarea x de tip a
  - Apoi efectuează comanda k x obținând o valoare y de tip b
  - Produce y ca rezultat al comenzii

Citește o linie!

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

Exemplu

Dat fiind șirul de intrare "abc\ndef", getLine produce șirul "abc" și șirul rămas de intrare e "def"

De la intrare la ieșire

```
echo :: IO ()
echo = getLine >>= \line ->
         if line == "" then
           return ()
         else
           putStrLn (map toUpper line) >>
           echo

main :: IO ()
main = echo
```

De la intrare la ieșire

```
echo :: IO ()
echo = getLine >>= \line ->
         if line == "" then
           return ()
         else
           putStrLn (map toUpper line) >>
           echo

main :: IO ()
main = echo
```

Test

```
Prelude> runghc Echo.hs
One line
ONE LINE
And, another line!
AND, ANOTHER LINE!
```

Notația „do”

Urmatoarele sunt echivalente:

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

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

Echo în notația „do”

```
echo :: IO ()
echo = getLine >>= \line ->
         if line == "" then
           return ()
         else
           putStrLn (map toUpper line) >>
           echo
```

Echivalent cu

```
echo :: IO ()
echo = do {
         line <- getLine;
         if line == "" then
           return ()
         else do {
           putStrLn (map toUpper line);
           echo
         }
```

Notația do pentru monade

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

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
```

```
do { x1 <- e1;
     e2;
     e3
   }
```

Notația „do” în general

- Fiecare linie x <– e; ... devine e >>= \x –> ...
- Fiecare linie e; ... devine e >> ...

De exemplu, urmatoarele sunt echivalente:

```
do { x1 <- e1;                e1   >>= \x1 ->
     x2 <- e2;                e2   >>= \x2 ->
     e3;                      e3   >>
     x4 <- e4;                e4   >>= \x4 ->
     e5;                      e5   >>
     e6                       e6
   }
```

Concluzii

Conținutul cursului de PF

- Elemente de baza
- Functii
- Liste
- Currying
- Operatori. Sectiuni
- Functii de nivel inalt
- Tipuri de date algebrice
- Clase de tipuri
- Functor
- Semigroup
- Monoid
- Foldable
- Applicative
- Monad

More to explore

- Testing; Property-Based Testing
- Random Numbers
- Monad Transformers
- Parallel and Concurrent Programming
- Free Monads
- Dependent Types

Haskell - best for

Application Domains	Common Programming Needs
Compilers	Maintenance
	Single-machine Concurrency
	Types / Type-driven development
	Parsing / Pretty-printing

Sursa https://github.com/Gabriel439/post-rfc/blob/main/sotu.md

Feedback

Seria 23: https://www.questionpro.com/t/AT4qgZqoGk

Seria 24: https://www.questionpro.com/t/AT4NiZqoGD

Seria 25: https://www.questionpro.com/t/AT4qgZqoGq

Mulțumim că ați participat la acest curs!

Multă baftă la examen!