Programare funcțională

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Sintaxă

- Comentarii

```haskell
-- comentariu pe o linie
{-  comentariu pe
    mai multe
    linii -}
```

- Identificatori
  - șiruri formate din litere, cifre, caracterele _ și ' (apostrof)
  - identificatorii pentru variabile încep cu literă mică sau _
  - identificatorii pentru tipuri și constructori încep cu literă mare
  - Haskell este sensibil la majuscule (case sensitive)

```haskell
double x =  2 * x
data Point a = Pt a a
```

Sintaxă

Blocuri și indentare

Blocurile sunt delimitate prin indentare.

```
fact n =  if n == 0
             then 1
             else  n * fact (n-1)

trei =  let
             a = 1
             b = 2
        in a + b
```

- echivalent, putem scrie

```
trei  =  let {a = 1; b = 2} in a + b
trei  =  let a = 1; b = 2 in a + b
```

Variabile

Presupunem că fisierul `test.hs` conține

```
x=1
x=2
```

- Ce valoare are x?

```
Prelude> :l test.hs

test.hs:2:1: error:
    Multiple declarations of ‘x’
    Declared at: test.hs:1:1
                 test.hs:2:1
  |
2 | x=2
  | ^
```

Variabile

În Haskell, variabilele sunt imuabile, adică:

- `=` **nu** este operator de atribuire
- `x = 1` reprezintă o *legatură* (binding)
- din momentul în care o variabilă este legată la o valoare, acea valoare nu mai poate fi schimbată

Legarea variabilelor

let .. in ...

este o *expresie* care crează scop local

Presupunem că fișierul `testlet.hs` conține

```
x=1
z= let x=3 in x
```

```
Prelude> :l testlet.hs
[1 of 1] Compiling Main
Ok, 1 module loaded.
*Main> z
3
*Main> x
1
```

Legarea variabilelor

- **let .. in ...** crează scop local

```
x = let
        z = 5
        g u = z + u                      -- x=12
    in  let
            z = 7
        in g 0 + z
```

```
x = let z = 5; g u = z + u in let z = 7 in g 0    -- x=5
```

- clauza **... where ...** creaza scop local

```
f x = g x + g x + z
  where
    g x = 2*x
    z =  x-1
```

Legarea variabilelor

- **let .. in ...** este o expresie

```
x = [let y =8 in y, 9] -- x=[8,9]
```

- **where** este o clauză, disponibilă doar la nivel de definiție

  x = [y where y =8, 9] – error: parse error ...

- Variabile pot fi legate și prin "pattern matching" la definirea unei funcții sau expresii **case**.

```
h x | x == 0    =  0
    | x == 1    = y + 1
    | x == 2    = y * y
    | otherwise = y
  where y = x*x
```

```
f x = case x of
              0 ->  0
              1 -> y + 1
              2 -> y * y
              _ -> y
  where y = x*x
```

Sistemul tipurilor

"There are three interesting aspects to types in Haskell: they are strong, they are static, and they can be automatically inferred."
http://book.realworldhaskell.org/read/types-and-functions.html

tare garanteaza absenta anumitor erori

static tipul fiecari valori este calculat la compilare

dedus automat compilatorul deduce automat tipul fiecarei expresii

```
Prelude> :t [('a',1,"abc")]
[('a',1,"abc")] :: Num b => [(Char, b, [Char])]
```

Sistemul tipurilor

Tipurile de baza

Int, Integer, Float, Double, Bool, Char, String

- tipuri compuse: tupluri si liste

```
Prelude> :t :t ('a', True)
('a', True) :: (Char, Bool)

Prelude> :t ["ana", "ion"]
["ana", "ion"] :: [[Char]]
```

- tipuri noi definite de utilizator

```
data RGB = Rosu | Verde | Albastru
data Point a = Pt a a      -- tip parametrizat
                           -- a este variabila de tip
```

Tipuri de date

- Integer: 4, 0, -5

```
Prelude> 4 + 3
Prelude> (+) 4 3
```

```
Prelude> mod 4 3
Prelude> 4 `mod` 3
```

- Float: 3.14

```
Prelude> truncate 3.14
Prelude> sqrt 4
```

```
Prelude> let x = 4 :: Int
Prelude> sqrt (fromIntegral x)
```

- Char: 'a','A', '\n'

```
Prelude> import Data.Char
Prelude Data.Char> chr 65
Prelude Data.Char> ord 'A'
Prelude Data.Char> toUpper 'a'
Prelude Data.Char> digitToInt '4'
```

Tipuri de date

- Bool: True, False

```
data Bool = True | False
```

```
Prelude> True && False || True
Prelude> not True
```

```
Prelude> 1 /= 2
Prelude> 1 == 2
```

- String: "prog\ndec"

```
type String = [Char] -- sinonim pentru tip
```

```
Prelude> "aa"++"bb"
"aabb"
Prelude> "aabb" !! 2
'b'
```

```
Prelude> lines "prog\ndec"
["prog","dec"]
Prelude> words "pr og\nde cl"
["pr","og","de","cl"]
```

Liste

Definiție

O

rice listă poate fi scrisă folosind doar constructorul (:) și lista vidă []

- [1,2,3] == 1 : (2 : (3 : [])) == 1 : 2 : 3 : []
- "abcd" == ['a','b','c','d'] == 'a' : ('b' : ('c' : ('d' : []))) == 'a' : 'b' : 'c' : 'd' : []

Definiție recursivă

O listă este

- vidă, notată []; sau
- compusă, notată x:xs, dintr-un un element x numit capul listei (head) și o listă xs numită coada listei (tail).

Tipuri de date compuse

- Tipul listă

```
Prelude>:t [True, False, True]
[True, False, True] :: [Bool]
```

- Tipul tuplu - secvențe de de tipuri deja existente

```
Prelude> :t (1 :: Int, 'a', "ab")
(1 :: Int, 'a', "ab") :: (Int, Char, [Char])

Prelude> fst (1,'a') -- numai pentru perechi
Prelude> snd (1,'a')
```

- Tipul unit

```
Prelude> :t ()
() :: ()
```

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim in GHCi dacă introducem comanda

```
Prelude> :t 1
```

Răspunsul primit este:

```
1 :: Num a => a
```

Semnificația este următoarea:

- a este un *parametru de tip*
- Num este o clasă de tipuri
- 1 este o valoare de tipul a din clasa Num

```
Prelude> :t 1
1 :: Num a => a

Prelude> :t [1,2,3]
[1,2,3] :: Num t => [t]
```

Funcții în Haskell. Terminologie

Prototipul funcției — **double** :: Integer -> Integer

- **numele funcției**
- signatura funcției

Definiția funcției — **double** elem = elem + elem

- **numele funcției**
- parametrul formal
- corpul funcției

Aplicarea funcției — **double** 5

- **numele funcției**
- parametrul actual (argumentul)

Exemplu: funcție cu **un** argument de tip tuplu

Prototipul funcției — **dist** :: (Integer, Integer) -> Integer

- **numele funcției**
- signatura funcției

Definiția funcției — **dist** (elem1, elem2) = abs (elem1 - elem2)

- **numele funcției**
- parametrul formal
- corpul funcției

Aplicarea funcției — **dist** (elem1, elem2)

- **numele funcției**
- argumentul

Tipuri de funcții

```
Prelude> :t abs
abs :: Num a => a -> a
Prelude> :t div
div :: Integral a => a -> a -> a

Prelude> :t (:)
(:) :: a -> [a] -> [a]
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]


Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

Definirea funcțiilor

```
fact :: Integer -> Integer
```

- Definiție folosind **if**

```
fact n = if n == 0 then 1
         else n * fact(n-1)
```

- Definiție folosind ecuații

```
fact 0 = 1
fact n = n * fact(n-1)
```

- Definiție folosind cazuri

```
fact n
  | n == 0    = 1
  | otherwise = n * fact(n-1)
```

Programare funcțională

Liste și funcții în Haskell

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Sistemul tipurilor

Tipurile de baza

Int, Integer, Float, Double, Bool, Char, String

- tipuri compuse: tupluri si liste

```
Prelude> :t :t ('a', True)
('a', True) :: (Char, Bool)

Prelude> :t ["ana", "ion"]
["ana", "ion"] :: [[Char]]
```

- tipuri noi definite de utilizator

```
data RGB = Rosu | Verde | Albastru
data Point a = Pt a a     -- tip parametrizat
                          -- a este variabila de tip
```

Liste

Definiție

Observație

Orice listă poate fi scrisă folosind doar constructorul (:) și lista vidă []

- [1,2,3] == 1 : (2 : (3 : [])) == 1 : 2 : 3 : []
- "abcd" == ['a','b','c','d'] == 'a' : ('b' : ('c' : ('d' : []))) == 'a' : 'b' : 'c' : 'd' : []

Definiție recursivă

O listă este

- vidă, notată []; sau
- compusă, notată x:xs, dintr-un un element x numit capul listei (head) și o listă xs numită coada listei (tail).

Definirea listelor. Operații

Intervale și progresii

```
interval = ['c'..'e']           -- ['c','d','e']
progresie = [20,17..1]          -- [20,17,14,11,8,5,2]
progresie' = [2.0,2.5..4.0] -- [2.0,2.5,3.0,3.5,4.0]
```

Operații

```
Prelude> [1,2,3] !! 2
3
Prelude> "abcd" !! 0
'a'
Prelude> [1,2] ++ [3]
[1,2,3]
Prelude> import Data.List
```

String = listă de caractere

- String: "prog\nfunc"

```
type String = [Char] -- sinonim pentru tip

Prelude> "aa"++"bb"
"aabb"
Prelude> "aabb" !! 2
'b'

Prelude> lines "prog\nfunc"
["prog","func"]
Prelude> words "pr og\nfu nc"
["pr","og","fu","nc"]
```

Definiția prin selecție $\{x \mid P(x)\}$

[E(x)| x <- [x1,. . . ,xn], P(x)]

```
Prelude> let xs = [0..10]
Prelude> [x | x <- xs, even x]
[0,2,4,6,8,10]

Prelude> let xs = [0..6]
Prelude> [(x,y) | x <- xs, y <- xs,  x + y == 10]
[(4,6),(5,5),(6,4)]
```

Folosirea lui `let` pentru declarații locale:

```
Prelude> [(i,j) | i <- [1..2], let k = 2 * i, j <- [1..k]]
[(1,1),(1,2),(2,1),(2,2),(2,3),(2,4)]

Prelude> let xs = ['A'..'Z']
Prelude> [x | (i,x) <- [1..] `zip` xs, even i]
"BDFHJLNPRTVXZ"
```

zip xs ys

```
Prelude> let xs = ['A'..'Z']
Prelude> [x | (i,x) <- [1..] `zip` xs, even i]
"BDFHJLNPRTVXZ"

Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]

Prelude> let ys = ['A'..'E']
Prelude> zip [1..]  ys
[(1,'A'),(2,'B'),(3,'C'),(4,'D'),(5,'E')]
```

Observați diferența!

```
Prelude> zip [1..3] ['A'..'D']
[(1,'A'),(2,'B'),(3,'C')]

Prelude> [(x,y) | x <- [1..3], y <- ['A'..'D']]
[(1,'A'),(1,'B'),(1,'C'),(1,'D'),(2,'A'),(2,'B'),(2,'C')
    ,(2,'D'),(3,'A'),(3,'B'),(3,'C'),(3,'D')]
```

Lenevire (Lazyness)

Argumentele sunt evaluate doar când e necesar și doar cât e necesar

```
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> let x = head []
Prelude> let f a = 5
Prelude> f x
5
Prelude> [1,head [],3] !! 0
1
Prelude> [head [],3] !! 1
*** Exception: Prelude.head: empty list
```

Liste infinite

Drept consecință a evaluării leneșe, se pot defini liste infinite (fluxuri de date)

```
Prelude> let natural = [0,..]
Prelude> take 5 natural
[0,1,2,3,4]

Prelude> let evenNat = [0,2..] -- progresie infinita
Prelude> take 7 evenNat
[0,2,4,6,8,10,12]

Prelude> let ones = [1,1..]
Prelude> let zeros = [0,0..]
Prelude> let both = zip ones zeros
Prelude> take 5 both
[(1,0),(1,0),(1,0),(1,0),(1,0)]
```

Exemplu: funcție cu **un** argument de tip tuplu

Prototipul funcției **dist** :: (Integer, Integer) -> Integer

- **numele funcției**
- signatura funcției

Definiția funcției **dist** (elem1, elem2) = abs (elem1 - elem2)

- **numele funcției**
- parametrul formal
- corpul funcției

Aplicarea funcției **dist** (elem1, elem2)

- **numele funcției**
- argumentul

Tipuri de funcții

```
Prelude> :t abs
abs :: Num a => a -> a

Prelude> :t div
div :: Integral a => a -> a -> a

Prelude> :t (:)
(:) :: a -> [a] -> [a]
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]


Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

Definirea funcțiilor folosind **if**

- analiza cazurilor folosind expresia "if"

```
semn : Integer -> Integer
semn n = if n < 0 then (-1)
         else if n=0 then 0
              else 1
```

- definiție recursivă în care analiza cazurilor folosește expresia "if"

```
fact :: Integer -> Integer
fact n = if n == 0 then 1
         else n * fact(n-1)
```

Definirea funcțiilor folosind **gărzi**

Funcția *semn* o putem defini astfel

$$semn\ n = \begin{cases} -1, & \textbf{dacă n} < \textbf{0} \\ 0, & \textbf{dacă n} = \textbf{0} \\ 1, & \textbf{altfel} \end{cases}$$

În Haskell, condițiile devin gărzi:

```
semn  n
    | n < 0       =  -1
    | n = 0       =   0
    | otherwise   =  1
```

Definirea funcțiilor folosind **gărzi**

Funcția *fact* o putem defini astfel

$$fact\ \ n = \begin{cases} 1, & \textbf{dacă n} = \mathbf{0} \\ n * fact(n-1), & \textbf{altfel} \end{cases}$$

În Haskell, condițiile devin gărzi:

```
fact n
    | n == 0     = 1
    | otherwise  = n * fact(n-1)
```

Definirea funcțiilor folosind șabloane și ecuații

```
semn :: Integer -> Integer
semn 0 = 0
semn x
  | x > 0     = 1
  | otherwise = -1

fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact(n-1)
```

- variabilele și valorile din partea stângă a semnului = sunt *șabloane*;
- când funcția este aplelată se încearcă potrivarea parametrilor actuali cu șabloanele, ecuațiile fiind încercate *în ordinea scrierii*;
- în definiția factorialului, 0 și n sunt șabloane: 0 se va potrivi numai cu el însuși, iar n se va potrivi cu orice valoare de tip `Integer`.

Definirea funcțiilor folosind șabloane și ecuații

- în Haskell, ordinea ecuațiilor este importantă

Să presupunem că schimbăm ordinii ecuațiilor din definiția factorialului:

```
fact :: Integer -> Integer
fact n = n * fact(n-1)
fact 0 = 1
```

Ce se întâmplă?

Deoarece n este un pattern care se potrivește cu orice valoare, inclusiv cu 0, orice apel al funcției va alege prima ecuație. Astfel, funcția nu își va încheia execuția pentru valori pozitive.

Definirea funcțiilor folosind șabloane și ecuații

Tipul Bool este definit în Haskell astfel:

```
data Bool = True | False
```

Putem defini operația || astfel

```
(||) :: Bool -> Bool -> Bool

False || x = x
True  || _ = True
```

În acest exemplu șabloanele sunt _, **True** și **False**.

Observăm că **True** și **False** sunt constructori de date și se vor potrivi numai cu ei înșiși.

Șablonul _ se numește *wild-card pattern*; el se potrivește cu orice valoare.

Șabloane pentru tupluri

Observați că (,) este constructorul pentru perechi.

```
(u,v)=('a',[(1,'a'),(2,'b')])  -- u='a',
                               -- v=[(1,'a'),(2,'b')]
```

- Definiții folosind șabloane

```
selectie :: Integer -> String -> String
```

```
-- case ... of
selectie x s =
    case (x,s) of
        (0,_) -> s
        (1, z:zs) -> zs
        (1, []) -> []
        _ -> (s ++ s)
```

```
-- stil ecuational
selectie 0 s = s
selectie 1 (_:s) = s
selectie 1 "" = ""
selectie _ s = s + s
```

Șabloane (patterns) pentru liste

Listele sunt construite folosind constructorii (:) și []

```
[1,2,3] == 1:[2,3]   -- == 1:2:[3] == 1:2:3:[]
```

Observați:

```
Prelude> let x:y = [1,2,3]
Prelude> x
1
Prelude> y
[2,3]
```

Ce s-a întâmplat?

- x:y este un șablon pentru liste
- potrivirea dintre x:y și [1,2,3] a avut ca efect:
  - "deconstrucția" valorii [1,2,3] în 1:[2,3]
  - legarea lui x la 1 și a lui y la [2,3]

Șabloane (patterns) pentru liste

Definiții folosind șabloane

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

- x:xs se potrivește cu liste nevide

Atenție!

Șabloanele sunt definite folosind constructori. De exemplu, operația de concatenare pe liste este (++) :: [a]-> [a] -> [a] dar
[x] ++ [1] = [2,1] nu va avea ca efect legarea lui x la 2;
încercând să evaluăm x vom obține un mesaj de eroare:

```
Prelude> [x] ++ [1] = [2,1]
Prelude> x
```

error: ...

Șabloanele sunt liniare

În Haskell șabloanele sunt liniare, adică o variabilă apare cel mult odată. Șabloane în care o variabilă apare de mai multe ori provoacă mesaje de eroare

```
x:x:[1] = [2,2,1]

ttail (x:x:t) = t

foo x x = x^2
```

error: Conflicting definitions for x

O soluție este folosirea gărzilor:

```
ttail (x:y:t) | (x==y) = t
              | otherwise = ...

foo x y | (x == y) = x^2
        | otherwise = ...
```

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.

- Pentru $x \in A$ (arbitrar, fixat) definim

  $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.

  Funcția f_x se obține prin aplicarea parțială a funcției f.

  In mod similar definim *aplicarea parțială* pentru orice $y \in B$

  $f^y : A \to C$, $f^y(x) = z$ dacă și numai dacă $f(x, y) = z$.

Funcții în matematică

Exemplu

A =Int, $B = C$ =String

$$f(x,y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

- Fie $x \in$ Int arbitrar, fixat. Atunci f_x :String→String și
  - dacă $x <= 0$, atunci $f_x(y) = ""$ oricare y
  - dacă $x > 0$ atunci $f_x(y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \end{cases}$
- Fie $y \in$String arbitrar, fixat. Atunci f^y :Int→String și

$$f^y(x) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.
- Pentru $x \in A$ (arbitrar, fixat) definim $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.
- Dacă notăm $B \to C \overset{not}{=} \{h : B \to C \mid h \text{ funcție}\}$ observăm că $f_x \in B \to C$ pentru orice $x \in A$.
- Asociem lui f funcția

  $$cf : A \to (B \to C), \;\; cf(x) = f_x$$

  Observăm că pentru fiecare element $x \in A$, funcția cf întoarce ca rezultat funcția $f_x \in B \to C$, adică

  $$cf(x)(y) = z \text{ dacă și numai dacă } f(x, y) = z$$

Forma **curry**

Vom spune că funcția cf este *forma curry* a funcției f.

De la matematică la Haskell

Funcția f : Int $\times$ String $\rightarrow$ String

$$f(x,y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

poate fi definită în Haskell astfel:

```
f :: (Int, String) -> String
f (n,s) = take n s
```

Observăm că:

```
Prelude> let cf = curry f
Prelude> :t cf
cf :: Int -> String -> String
Prelude> f(1,"abc")
"a"
Prelude> cf 1 "abc"
"a"
```

Currying

"Currying" este procedeul prin care o funcție cu mai multe argumente este transformată într-o funcție care are un singur argument și întoarce o altă funcție.

- In Haskell toate funcțiile sunt forma **curry**, deci au un singur argument.

- Operatorul $\to$ pe tipuri este asociativ la dreapta, adică tipul $a_1 \to a_2 \to \cdots \to a_n$ îl gândim ca $a_1 \to (a_2 \to \cdots (a_{n-1} \to a_n) \cdots)$.
- Aplicarea funcțiilor este asociativă la stânga, adică expresia $f\ x_1 \cdots x_n$ o gândim ca $(\cdots((f\ x_1)\ x_2) \cdots x_n)$.

Funcții și mulțimi

Teoremă

Mulțimile $(A \times B) \to C$ și $A \to (B \to C)$ sunt echipotente.

Observație

Funcțiile curry și uncurry din Haskell stabilesc bijecția din teoremă:

```
Prelude> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
Prelude> :t uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c
```

Tipuri de funcții

Fie foo o funcție cu următorul tip

```
foo :: a -> b -> [a] -> [b]
```

- are trei argumente, de tipuri a, b și [a]
- întoarce un rezultat de tip [b]

Schimbăm signatura funcției astfel:

```
ffoo :: (a -> b) -> [a] -> [b]
```

- are două argumente, de tipuri (a -> b) și [a],
  adică o funcție de la a la b și o listă de elemente de tip a
- întoarce un rezultat de tip [b]

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```

Funcții anonime

Funcții anonime = lambda expresii

\x1 x2 · · · xn -> expresie

```
Prelude> (\x -> x + 1) 3
4
Prelude> inc = \x -> x + 1
Prelude> add = \x y -> x + y
Prelude> aplic = \f x -> f x

Prelude> map (\x -> x+1) [1,2,3,4]
[2,3,4,5]
```

- Funcțiile sunt valori (first-class citizens)
  - pot fi folosite ca argumente pentru alte funcții

Structura λ-expresiilor

O expresie este definită recursiv astfel:

- este o variabilă (un identificator)
  x
- se obține prin abstractizarea unei variabile x într-o altă expresie e
  $\lambda x.e$ exemplu: $\lambda x.x$
- se obține prin aplicarea unei expresii e_1 asupra alteia e_2
  $e_1\ e_2$ exemplu: $(\lambda x.x)y$

Operația de abstractizare $\lambda x.e$

- reprezintă o funcție anonimă
- constă din două parți: antetul $\lambda x.$ și corpul e
- variabila x din anter este parametrul funcției
  - leagă aparițiile variabilei x în e (ca un cuantificator)
  - Exemplu: $\lambda x.xy$ — x e legată, y e liberă
- Corpul funcției reprezintă expresia care definește funcția

α-echivalență

- Redenumirea unui argument și a tuturor aparițiilor sale legate
    - Exemplu: $\lambda x.x \equiv_\alpha \lambda y.y \equiv_\alpha \lambda a.a$
    - Asemanator cu: $f(x) = x$ vs $f(y) = y$ vs $f(a) = a$
- Numele asociat argumentului e pur formal
    - E necesar doar ca să îl pot recunoaște în corpul funcției
    - Există reprezentări fără argumente (e.g. indecși de Bruijn)
- α-echivalența redenumește doar aparițiile legate ale argumentului
    - Exemple:
      $(\lambda x.x)x \not\equiv_\alpha (\lambda y.y)y$
      $(\lambda x.x)x \equiv_\alpha (\lambda y.y)x$

β-reducție

Cum aplicăm o funcție (anonimă) unui argument?

înlocuim aplicația cu corpul funcției în care substituim aparițiile variabilei legate cu argumentul dat.

$$(\lambda x.e)e' \to_\beta e[x := e']$$

Exemple

$$(\lambda x.x)y \to_\beta x[x := y] = y$$

$$(\lambda x.x\ x)\lambda x.x \to_\beta x\ x[x := \lambda x.x] = (\lambda x.x)\lambda x.x \to_\beta x[x := \lambda x.x] = \lambda x.x$$

β-reducție — alte exemple

Aplicarea funcțiilor se grupează la stânga

$(\lambda x.x)(\lambda y.y)z = ((\lambda x.x)(\lambda y.y))z$
$((\lambda x.x)(\lambda y.y))z \rightarrow_\beta (x[x := \lambda y.y])z = (\lambda y.y)z$
$(\lambda y.y)z \rightarrow_\beta y[y := z] = z$

Funcție cu variabile libere

$(\lambda x.x\ y)z \rightarrow_\beta (x\ y[x := z]) = z\ y$

lambda are are prioritate foarte mică

$\lambda x.x\ \lambda x.x = \lambda x.(x\ (\lambda x.x))$

Mai multe argumente

- Funcțiile anonime au un singur parametru
  - și pot fi aplicate unui singur argument
- Simulăm mai multe argumente prin abstractizare repetată
  Exemplu: $\lambda x.\lambda y.x\ y$
  - Citim: primește ca argumente x și y și aplică pe x lui y
  - De fapt e o funcție de x care în urma aplicării întoarce cadă o funcție de y
  - Pentru simplificarea notației, scriem $\lambda x\ y.x\ y$ în loc de $\lambda x.\lambda y.x\ y$

Exemplu de evaluare

$(\lambda x\ y.x\ y)(\lambda z.a)1 = (\lambda x.(\lambda y.x\ y))(\lambda z.a)1 = ((\lambda x.(\lambda y.x\ y))(\lambda z.a))1 \rightarrow_\beta$
$((\lambda y.x\ y)[x := \lambda z.a])1 = (\lambda y.(\lambda z.a)y)1 \rightarrow_\beta ((\lambda z.a)y)[y := 1] =$
$(\lambda z.a)1 \rightarrow_\beta a[z := y] = a$

Programarea funcțională

- Paradigmă de programare ce folosește funcții modelate după funcțiile din matematică.
- Programele se obțin ca o combinație de expresii.
- Expresiile pot fi valori concrete, variable și funcții.
- Funcțiile sunt expresii ce pot fi aplicate unor intrări.
  - În urma aplicării, o funcție e redusă sau evaluată.
- Funcțiile sunt valori (first-class citizens)
  - pot fi folosite ca argumente pentru alte funcții

Programare funcțională

Funcții de ordin înalt. Procesarea fluxurilor de date.

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

20 octombrie 2020

Operatorii sunt funcții cu două argumente

Operatorii în Haskell

- au două argumente
- pot fi apelați folosind notația infix
- pot fi definiți folosind numai "simboluri" (ex: *!*)
  - în definiția tipului operatorul este scris între paranteze

- Operatori predefiniți

```haskell
(||) :: Bool -> Bool -> Bool
(:) :: a -> [a] -> [a]
(+) :: Num a => a -> a -> a
```

- Operatori definiți de utilizator

```haskell
(&&&) :: Bool -> Bool -> Bool  -- atentie la paranteze
True &&& b = b
False &&& _ = False
```

Funcții ca operatori

```
Prelude> mod 5 2
1
Prelude> 5 `mod` 2
1
```

- operatorii care sunt definiți în formă infix, sunt apelați în formă prefix folosind paranteze
  2 + 3 == (+) 2 3
- operatorii care sunt definiți în formă prefix, sunt apelați în formă infix folosind ‘ ‘
  mod 5 2 == 5 ‘mod‘ 2

```
elem :: a -> [a] -> Bool
Prelude> 1 `elem` [1,2,3]
True
```

Precedență și asociativitate

```
Prelude> 3+5*4:[6]++8-2+3:[2]==[23,6,9,2]||True==False
True
```

Precedență și asociativitate

```
Prelude> 3+5*4:[6]++8-2+3:[2]==[23,6,9,2]||True==False
True
```

Precedence	Left associative	Non-associative	Right associative
9	!!		.
8			^, ^^, **
7	*, /, ‘**div**‘, ‘**mod**‘, ‘**rem**‘, ‘**quot**‘		
6	+, −		
5			:, ++
4		==, /=, <, <=, >, >=, ‘**elem**‘, ‘**notElem**‘	
3			&&
2			||
1	>>, >>=		
0			$, $!, ‘**seq**‘

Asociativitate

Operatorul - asociativ la stanga

5 − 2 − 1 == (5 − 2) − 1 −− /= 5 − (2 − 1)

Operatorul : asociativ la dreapta

5 : 2 : [] == 5 : (2 : [])

Operatorul ++ asociativ la dreapta

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

l1 ++ l2 ++ l3 ++ l4 ++ l5 == l1 ++ (l2 ++ (l3 ++ (l4 ++ l5)))

Asociativitate

Operatorul - asociativ la stanga

$5 - 2 - 1 == (5 - 2) - 1$ -- /= $5 - (2 - 1)$

Operatorul : asociativ la dreapta

5 : 2 : [] == 5 : (2 : [])

Operatorul ++ asociativ la dreapta

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

l1 ++ l2 ++ l3 ++ l4 ++ l5 == l1 ++ (l2 ++ (l3 ++ (l4 ++ l5)))

Care este complexitatea aplicării operatorului ++?

- liniară în lungimea primului argument

Secțiuni ("operator sections")

Secțiunile operatorului binar op sunt (op e) și (e op).
Matematic, ele corespund aplicării parțiale a funcției op.

Aplicarea parțială

Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(a,b) = c$ unde $a \in A$, $b \in B$ și $c \in C$.

Pentru $a \in A$ și $b \in B$ (arbitrare, fixate) definim

$f_a : B \to C$, $f_a(b) = c$ dacă și numai dacă $f(a,b) = c$,

$f^b : A \to C$, $f^b(a) = c$ dacă și numai dacă $f(a,b) = c$.

Funcțiile f_a și f_b se obțin prin aplicarea parțială a funcției f.

Secțiuni ("operator sections")

Secțiunile operatorului binar op sunt (op e) și (e op).
Matematic, ele corespund aplicării parțiale a funcției op.

- secțiunile lui ++ sunt (++ e) și (e ++)

```
Prelude> :t (++ " world!")
(++ " world!") :: [Char] -> [Char]
Prelude> (++ " world!") "Hello"  -- atentie la
    paranteze
"Hello world!"
Prelude> ++ " world!" "Hello"
```

error

- secțiunile lui <-> sunt (<-> e) și (e <->), unde

```
Prelude> let x <-> y = x-y+1  -- definit de utilizator
Prelude> :t (<-> 3)
(<-> 3) :: Num a => a -> a
Prelude> (<-> 3) 4
2
```

Secțiuni

- Secțiunile operatorului (:)

```
Prelude> (2:)[1,2]
[2,1,2]
Prelude> (:[1,2]) 3
[3,1,2]
```

- Secțiunile sunt afectate de asociativitatea și precedența operatorilor.

```
Prelude> :t (+ 3 * 4)
(+ 3 * 4) :: Num a => a -> a

Prelude> :t (* 3 + 4)
error -- + are precedenta mai mica decat *
Prelude> :t (* 3 * 4)
error  -- * este asociativa la stanga
Prelude> :t (3 * 4 *)
(3 * 4 *) :: Num a => a -> a
```

Funcții anonime și secțiuni

Sectiunile sunt definite prin lambda expresii:

- (‘op‘ 2) e forma scurtă a lui (\x -> x ‘op‘ 2)
- (2 ‘op‘) e forma scurtă a lui (\x -> 2 ‘op‘ x)

Exemple

- (> 0) e forma scurtă a lui (\x -> x > 0)
- (2 *) e forma scurtă a lui (\x -> 2 * x)
- (+ 1) e forma scurtă a lui (\x -> x + 1)
- (2 ^) e forma scurtă a lui (\x -> 2 ^ x)
- (^ 2) e forma scurtă a lui (\x -> x ^ 2)

Compunerea funcțiilor — operatorul .

Matematic

Date fiind $f : A \to B$ și $g : B \to C$, compunerea lor, notată $g \circ f : A \to C$ este dată de formula

$$(g \circ f)(x) = g(f(x))$$

În Haskell

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

Exemplu

```
Prelude> :t reverse
reverse :: [a] -> [a]
Prelude> :t take
take :: Int -> [a] -> [a]
Prelude> :t take 5 . reverse
take 5 . reverse :: [a] -> [a]

Prelude> (take 5 . reverse) [1..10]
[10,9,8,7,6]

Prelude> (head . reverse . take 5) [1..10]
5
```

Operatorul $

Operatorul ($) are precedența 0.

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

```
Prelude> (head . reverse . take 5) [1..10]
5

Prelude> head . reverse . take 5 $ [1..10]
5
```

Operatorul ($) este asociativ la dreapta.

```
Prelude> head $ reverse $ take 5 $ [1..10]
5
```

Pătrate

Definiți o funcție care pentru o listă de numere întregi dată ridică la pătrat fiecare element din listă.

```
*Main> squares [1,-2,3]
[1,4,9]
```

Soluție descriptivă

```haskell
squares :: [Int] -> [Int]
squares xs = [ x * x | x <- xs ]
```

Soluție recursivă

```haskell
squares :: [Int] -> [Int]
squares []     = []
squares (x:xs) = x*x : squares xs
```

Coduri ASCII

Transformați un șir de caractere în lista codurilor ASCII ale caracterelor.

```
*Main> ords "a2c3"
[97,50,99,51]
```

Soluție descriptivă

```
ords :: [Char] -> [Int]
ords xs = [ ord x | x <- xs ]
```

Soluție recursivă

```
ords :: [Char] -> [Int]
ords []     = []
ords (x:xs) = ord x : ords xs
```

Funcția **map**

Definiție

Date fiind o funcție de transformare și o listă, aplicați funcția fiecărui element al unei liste date.

Soluție descriptivă

```haskell
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]
```

Soluție recursivă

```haskell
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Exemplu — Pătrate

Soluție descriptivă

```
squares :: [Int] -> [Int]
squares xs = [ x * x | x <- xs ]
```

Soluție recursivă

```
squares :: [Int] -> [Int]
squares []     = []
squares (x:xs) = x*x : squares xs
```

Soluție folosind **map**

```
squares :: [Int] -> [Int]
squares xs = map sqr xs
  where sqr x = x * x
```

Exemplu — Coduri ASCII

Soluție descriptivă

```haskell
ords :: [Char] -> [Int]
ords xs = [ ord x | x <- xs ]
```

Soluție recursivă

```haskell
ords :: [Char] -> [Int]
ords []     = []
ords (x:xs) = ord x : ords xs
```

Soluție folosind **map**

```haskell
ords :: [Char] -> [Int]
ords xs = map ord xs
```

Selectarea elementelor pozitive dintr-o listă

```
*Main> positives [1,-2,3]
[1,3]
```

Soluție descriptivă

```haskell
positives :: [Int] -> [Int]
positives xs = [ x | x <- xs, x > 0 ]
```

Soluție recursivă

```haskell
positives :: [Int] -> [Int]
positives []                    = []
positives (x:xs) | x > 0        = x : positives xs
                 | otherwise    = positives xs
```

Selectarea cifrelor dintr-un șir de caractere

```
*Main> digits "a2c3"
"23"
```

Soluție descriptivă

```
digits :: [Char] -> [Char]
digits xs = [ x | x <- xs, isDigit x ]
```

Soluție recursivă

```
digits :: [Char] -> [Char]
digits []                   = []
digits (x:xs) | isDigit x = x : digits xs
              | otherwise = digits xs
```

Funcția **filter**

Definiție

Date fiind un predicat (funcție booleană) și o listă, selectați elementele din listă care satisfac predicatul.

Soluție descriptivă

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]
```

Soluție recursivă

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p []                   = []
filter p (x:xs) | p x         = x : filter p xs
                | otherwise   = filter p xs
```

Exemplu — Pozitive

Soluție descriptivă

```haskell
positives :: [Int] -> [Int]
positives xs = [ x | x <- xs, x > 0 ]
```

Soluție recursivă

```haskell
positives :: [Int] -> [Int]
positives []                    = []
positives (x:xs) | x > 0        = x : positives xs
                 | otherwise    = positives xs
```

Soluție folosind **filter**

```haskell
positives :: [Int] -> [Int]
positives xs = filter pos xs
  where pos x = x > 0
```

Exemplu — Cifre

Soluție descriptivă

```
digits :: [Char] -> [Char]
digits xs = [ x | x <- xs, isDigit x ]
```

Soluție recursivă

```
digits :: [Char] -> [Char]
digits []                      = []
digits (x:xs) | isDigit x = x : digits xs
              | otherwise = digits xs
```

Soluție folosind **filter**

```
digits :: [Char] -> [Char]
digits xs = filter isDigit xs
```

Suma

Definiți o funcție care dată fiind o listă de numere întregi calculează suma elementelor din listă.

```
*Main> sum [1,2,3,4]
10
```

Soluție recursivă

```haskell
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Produs

Definiți o funcție care dată fiind o listă de numere întregi calculează produsul elementelor din listă.

```
*Main> product [1,2,3,4]
24
```

Soluție recursivă

```haskell
product :: [Int] -> Int
product []     = 1
product (x:xs) = x * sum xs
```

Concatenare

Definiți o funcție care concatenează o listă de liste.

```
*Main> concat [[1,2,3],[4,5]]
[1,2,3,4,5]

*Main> concat ["con","ca","te","na","re"]
"concatenare"
```

Soluție recursivă

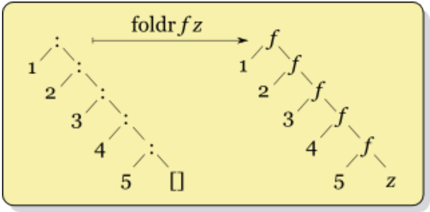
```
concat :: [[a]] -> [a]
concat []         = []
concat (xs:xss) = xs ++ concat xss
```

Funcția **foldr**

Definiție

foldr :: (a –> b –> b) –> b –> [a] –> b

Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare inițială, și o listă, calculați valoarea obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.



Suma

Soluție recursivă

```haskell
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Soluție folosind **foldr**

```haskell
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

Exemplu

```haskell
foldr (+) 0 [1, 2, 3] == 1 + (2 + (3 + 0))
```

Produs

Soluție recursivă

```
product :: [Int] -> Int
product []      = 1
product (x:xs) = x * sum xs
```

Soluție folosind **foldr**

```
product :: [Int] -> Int
product xs = foldr (*) 1 xs
```

Exemplu

```
foldr (*) 1 [1, 2, 3] == 1 * (2 * (3 * 1))
```

Concatenare

Soluție recursivă

```
concat :: [[a]] -> [a]
concat []         = []
concat (xs:xss) = xs ++ concat xss
```

Soluție folosind **foldr**

```
concat :: [Int] -> Int
concat xs = foldr (++) [] xs
```

Exemplu

```
foldr (++) [] ["Ana ", "are ", "mere."]
  == "Ana " ++ ("are " ++ ("mere." ++ []))
```

Suma pătratelor numerelor pozitive

```haskell
f :: [Int] -> Int
f xs = sum (squares (positives xs))

f :: [Int] -> Int
f xs = sum [ x*x | x <- xs, x > 0 ]

f :: [Int] -> Int
f []                   = 0
f (x:xs) | x > 0       = (x*x) + f xs
         | otherwise   = f xs

f :: [Int] -> Int
f xs = foldr (+) 0 (map sqr (filter pos xs))
  where
    sqr x = x * x
    pos x = x > 0
```

Foldr cu secțiuni — Exemplu

Folosind λ-expresii

```
f :: [Int] -> Int
f xs = foldr (+) 0
         (map (\ x -> x * x)
            (filter (\ x -> x > 0) xs))
```

Folosind secțiuni

```
f :: [Int] -> Int
f xs = foldr (+) 0 (map (^ 2) (filter (> 0) xs))
```

Operatorul . — stilul funcțional

Definiție cu parametru explicit

```
f :: [Int] -> Int
f xs = foldr (+) 0 (map (^ 2) (filter (> 0) xs))
```

Definiție compozițională

```
f :: [Int] -> Int
f = foldr (+) 0 . map (^ 2) . filter (> 0)
```

Map/Filter/Reduce în Haskell

Problemă

Aflați lungimea celui mai lung cuvânt care începe cu litera 'c' dintr-o listă dată.

```
strs = ["cezara", "petru", "claudia", "", "virgil"];
maxLengthFn = foldr max 0 .
              map length .
              filter testC
  where testC ('c':_) = True
        testC _       = False
maxLength = maxLengthFn strs
```

Map/Filter/Reduce în Python

http://www.python-course.eu/lambda.php

```
strs = ["cezara", "petru", "claudia", "", "virgil"];
def maxLengthFn(strs):
  return reduce(max,
          map(len,
            filter(lambda s: len(s) > 0 and s[0] == 'c',
              strs)));
maxLength = maxLengthFn(strs);
print(maxLength);
```

Programare funcțională

Procesarea fluxurilor de date. Evaluare leneșă.

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

3 noiembrie 2020

Funcțiile sunt valori

Funcțiile sunt valori!

```
Prelude> ap n f = if (n<=0) then id else (f . (ap (n-1) f))

Prelude> ap 3 (\x -> x*x) 4
65536

Prelude> ap 3 (\ (x, y) -> (x*x, y+y)) (4,5)
(65536,40)
```

Observați folosirea funcțiilor anonime (λ-expresii)!

```
Prelude> :t ap
ap :: (Eq t, Num t) => t -> (b -> b) -> b -> b

Prelude> :t ap 5
ap 5 :: (b -> b) -> b -> b
```

Funcțiile sunt valori

```
Prelude> ap n f = if (n<=0) then id else (f . (ap (n-1) f))

Prelude> :t ap
ap :: (Eq t, Num t) => t -> (b -> b) -> b -> b

Prelude> g = ap 2 (\x -> x*x)
Prelude> g 3 == ap 2 (\x -> x*x) 3
True

Prelude> g  == ap 2 (\x -> x*x)
error
```

Funcțiile nu pot fi comparate!

Din nou **map**

map :: (a -> b) -> [a] -> [b]

```
map f xs = [f x | x <- xs]
```

$$\begin{array}{ccccccccc} x_1 & : & x_2 & : & \cdots & : & x_n & : & [\,] \\ \downarrow & & \downarrow & & & & \downarrow & & \\ f(x_1) & : & f(x_2) & : & \cdots & : & f(x_n) & : & [\,] \end{array}$$

Problemă

Scrieți o funcție care scrie un șir de caractere cu litere mari.

```
scrieLitereMari s = map toUpper s
```

```
Prelude Data.Char> :t toUpper
toUpper :: Char -> Char
```

```
Prelude Data.Char> map toUpper "abac"
"ABAC"
```

Din nou **filter**

filter :: (a -> Bool) -> [a] -> [a]

```
filter prop xs  = [x | x <- xs, prop x]
```

```
Prelude> filter (>= 2) [1,3,4]
[3,4]
Prelude> filter (\ (x,y)-> x+y >= 10) [(1,4),(2,7), (3,10)]
[(3,10)]
```

Problemă

Scrieți o funcție care scrie selectează dintr-o listă de cuvinte pe cele care încep cu literă mare.

```
incepeLM xs = filter (\x -> isUpper (head x))  xs
```

```
Prelude Data.Char> incepeLM ["carte","Ana","minge","Petre"]
["Ana","Petre"]
```

map și filter

http://learnyouahaskell.com/higher-order-functions

Secvență Collatz: $c_1, c_2, \ldots, c_n$ (numere naturale)

$$x_{n+1} = \begin{cases} x_n/2 & \text{dacă } x_n \text{ este par} \\ 3x_n + 1 & \text{dacă } x_n \text{ este impar} \end{cases}$$

Exemplu: 22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1

Conjectura lui Collatz:

orice secvență Collatz se termină cu 1

Problemă

1. Scrieți o funcție care calculează secvența lui Collatz care începe cu n.
2. Determinați secvențele Collatz de lungime ≤ 15 care încep cu un număr din intervalul $[1, 100]$

Secvență Collatz

Secvență Collatz: $c_1, c_2, \ldots, c_n$ (numere naturale)

$$x_{n+1} = \begin{cases} x_n/2 & \text{dacă } x_n \text{ este par} \\ 3x_n + 1 & \text{dacă } x_n \text{ este impar} \end{cases}$$

Problemă

1. Scrieți o funcție care calculează secvența lui Collatz care începe cu n

```
collatz n
  | n == 1 = []
  | n  > 1 = n : collatz (next n)
  where next x | even x     = x `div` 2
               | otherwise = 3 * x + 1
```

map și filter

Problemă

1. Scrieți o funcție care calculează secvența lui Collatz care începe cu *n*.

```
collatz n
  | n == 1 = []
  | n  > 1 = n : collatz (next n)
  where next x | even x      = x `div` 2
               | otherwise = 3 * x + 1)
```

2. Determinați secvențele Collatz de lungime ≤ 5 care încep cu un număr din intervalul $[1, 100]$.

```
Prelude> filter (\x -> length x <= 5) (map collatz [1..100])
```

```
[[1],[2,1],[4,2,1],[8,4,2,1],[16,8,4,2,1]]
```

Funcții de ordin înalt

foldr și foldl

Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare inițială, și o listă, calculați valoare obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.

foldr :: (a -> b -> b) -> b -> [a] -> b

```
foldr op z [a1, a2, a3, ..., an] =
      a1 `op` (a2 `op` (a3 `op` (... (an `op` z) ...)))
```

foldl :: (b -> a -> b) -> b -> [a] -> b

```
foldl op z [a1, a2, a3, ..., an] =
      ( ... (((z `op` a1) `op` a2) `op` a3) ...) `op` an
```

foldr și foldl

Definiție

Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare inițială, și o listă, calculați valoarea obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.

Funcția *foldr*

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f i []     = i
foldr f i (x:xs) = f x (foldr f i xs)
```

Funcția *foldl*

```
foldl :: (b -> a-> b) -> b -> [a] -> b
foldl h i []     = i
foldl h i (x:xs) = foldl h (h i  x) xs
```

Filtrare, transformare, agregare

Problemă

Aflați lungimea celui mai lung cuvânt care începe cu litera 'c' dintr-o listă dată.

Definiția compozițională:

```
maxLengthFn = foldr max 0 .
              map length .
              filter (\x -> head x =='c')
```

Proprietatea de universalitate

Observație

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f i :: [a] -> b
```

Teoremă

Fie g o funcție care procesează liste finite. Atunci

$$\begin{array}{lcl} g\,[] & = & i \\ g\,(x : xs) & = & f\ x\ (g\ xs) \end{array} \quad \Leftrightarrow \quad g = \mathit{foldr}\ f\ i$$

Teorema determină condiții necesare și suficiente pentru ca o funcție g care procesează liste să poată fi definită folosind **foldr**.

Compunerea funcțiilor

În definiția lui **foldr**

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

b poate fi tipul unei funcții.

```
compose :: [a -> a] -> (a -> a)
compose = foldr (.) id
```

```
Prelude> foldr (.) id [(+1), (^2)] 3
10

-- functia (foldr (.) id [(+1), (^2)]) aplicata lui 3
```

Suma

Definiți o funcție care dată fiind o listă de numere întregi calculează suma elementelor din listă.

Soluție cu **foldr**

```
sum = foldr (+) 0
```

În definiția de mai sus elementele sunt procesate de la dreapta la stânga:
$\mathbf{sum}[x_1, \ldots, x_n] = (x_1 + (x_2 + \ldots (x_n + 0) \ldots)$

Problemă

Scrieți o definiție a sumei folosind **foldr** astfel încât elementele să fie procesate de la stânga la dreapta.

Suma

sum cu acumulator

```
sum :: [Int] -> Int
sum  xs = suml xs 0
          where
             suml [] n =n
             suml (x:xs) n = suml xs (n+x)
```

În definiția de mai sus elementele sunt procesate de la stânga la dreapta:
suml $[x_1, \ldots, x_n]\ 0 = (\ldots(0 + x_1) + x_2) + \ldots x_n)$
Definim suml cu **foldr**

- Obervăm că

```
suml :: [Int] -> (Int -> Int)
```

- Definim suml cu **foldr** aplicând proprietatea de universalitate.

Definirea suml cu foldr

Proprietatea de universalitate

$$\begin{array}{lcl} g\ [] & = & i \\ g\ (x : xs) & = & f\ x\ (g\ xs) \end{array} \quad \Leftrightarrow \quad g = foldr\ f\ i$$

Observăm că

```
suml  []  =  id      -- suml  []  n  =  n
```

Vrem să găsim *f* astfel încât

$$suml\ (x : xs) = f\ x\ (suml\ xs)$$

deoarece, din proprietatea de universalitate, va rezulta că

$$suml = foldr\ f\ id$$

Definirea suml cu foldr

```
suml :: [Int] -> (Int -> Int)
```

$$\begin{array}{lcll} suml\ (x : xs) & = & f\ x\ (suml\ xs) & \text{(vrem)} \\ suml\ (x : xs)\ n & = & f\ x\ (suml\ xs)\ n & \text{(vrem)} \\ suml\ xs\ (n + x) & = & f\ x\ (suml\ xs)\ n & (\text{def } suml) \end{array}$$

Notăm $u = suml\ xs$ și obținem

$$u\ (n + x) \quad = \quad f\ x\ u\ n$$

Soluție

```
f = \ x u n -> u (n+x)

suml = foldr (\ x u -> f x u) id

suml = foldr (\x u -> (\n -> u (n+x))) id

suml = foldr (\x u n -> u (n+x)) id
       -- tipurile sunt determinate corespunzator
```

Definirea sum cu foldr

```haskell
sum :: [Int] -> Int

sum xs = foldr (\ x u n -> u (n+x)) id xs 0

-- sum xs = suml xs 0
```

```
Prelude> sum xs = foldr (\ x u -> \ n -> u (n+x)) id xs 0
Prelude> sum [1,2,3]
6
```

Inversarea elementelor unei liste

Definiți o funcție care dată fiind o listă de elemente, calculează lista în care elementele sunt scrise în ordine inversă.

Soluție cu **foldl**

```
rev = foldl (<:>) []
  where (<:>) = flip (:)   -- flip (:) :: [a] -> a -> [a]
```

În definiția de mai sus elementele sunt procesate de la stânga la dreapta:
$\textbf{rev}[x_1, \ldots, x_n] = (...(([] <:> x_1) <:> x_2).....) <:> x_n$

Problemă

Scrieți o definiție a funcției rev folosind **foldr**.

Inversarea elementelor unei liste

rev cu acumulator

```
rev :: [a] -> [a]
rev xs = revl xs []
  where
    revl [] l        = l
    revl (x:xs) rxs = revl xs (x:rxs)
```

În definiția de mai sus elementele sunt procesate de la stânga la dreapta, fiind mutate în argumentul auxiliar:

Definim suml cu **foldr**

- Obervăm că

  ```
  revl :: [a] -> ([a] -> [a])
  ```

- Definim revl cu **foldr** aplicând proprietatea de universalitate.

Definirea revl cu foldr

Proprietatea de universalitate

$$\begin{array}{lcl} g\ [] & = & i \\ g\ (x : xs) & = & f\ x\ (g\ xs) \end{array} \quad \Leftrightarrow \quad g = foldr\ f\ i$$

Observăm că

```
revl [] = id     -- revl [] l = l
```

Vrem să găsim f astfel încât

$$revl\ (x : xs) = f\ x\ (revl\ xs)$$

deoarece, din proprietatea de universalitate, va rezulta că

$$revl = foldr\ f\ id$$

Definirea revl cu foldr

```
revl :: [a] -> ([a] -> [a])
```

$$\begin{array}{lcll} \mathit{revl}\ (x : xs) & = & f\ x\ (\mathit{revl}\ xs) & \text{(vrem)} \\ \mathit{revl}\ (x : xs)\ xs' & = & f\ x\ (\mathit{revl}\ xs)\ xs' & \text{(vrem)} \\ \mathit{revl}\ xs\ (x : xs') & = & f\ x\ (\mathit{revl}\ xs)\ xs' & \text{(def } \mathit{revl}) \end{array}$$

Notăm $u = \mathit{revl}\ xs$ și obținem

$$u\ (x : xs') \quad = \quad f\ x\ u\ xs'$$

Soluție

```
f = \ x u xs' -> u (x:xs')

revl = foldr (\ x u -> f x u) id

revl = foldr (\x u -> (\xs' -> u (x:xs'))) id

revl = foldr (\x u n -> u (x:xs')) id
        -- tipurile sunt determinate corespunzator
```

Definirea rev cu foldr

```haskell
rev :: [a] -> [a]

rev xs = foldr (\ x u xs' -> u (x:xs')) id xs []

-- rev xs = revl xs []
```

```
Prelude> rev xs = foldr (\ x u xs' -> u (x:xs')) id xs []
Prelude> rev [1,2,3]
[3,2,1]
```

foldl

Definiție

Funcția *foldl*

```
foldl :: (b -> a-> b) -> b -> [a] -> b
foldl h i []     = i
foldl h i (x:xs) = foldl h (h i  x) xs
```

```
foldl h i xs = foldl' h xs i
        where
          foldl' h [] i = i
          foldl' h (x:xs) i = foldl' h xs (h i x)
```

```
foldl' :: (b -> a -> b) -> [a] -> b -> b
foldl' h :: [a] -> (b ->b)
foldl' h xs :: b -> b
```

foldl’ cu **foldr**

Observăm că

```haskell
foldl' h [] = id    -- suml [] n = n
```

Vrem să găsim f astfel încât

$$foldl'\ h\ (x : xs) = f\ x\ (foldl'\ h\ xs)$$

deoarece, din proprietatea de universalitate, va rezulta că

$$foldl'\ h = foldr\ f\ id$$

foldl cu foldr

Soluție

```
h :: b -> a -> b

foldl' h = foldr f id

f = \ x u -> \ y -> u (h y x)


foldl h i xs = foldl' h xs i

foldl h i xs = foldr (\ x u -> \ y -> u (h y x)) id xs i
```

foldl cu foldr

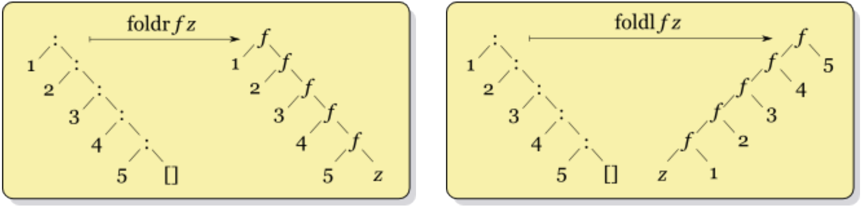
```
Prelude> let myfoldl h i xs =
                    foldr (\x u -> \y -> u (h y x)) id xs i

Prelude> myfoldl (+) 0 [1,2,3]
6

Prelude> let sing = (:[])  -- sing x = [x]

Prelude> take 3 (foldr (++) [] (map sing [1..]))
[1,2,3]

Prelude> take 3 (myfoldl (++) [] (map sing [1..]))
Interrupted.

Prelude> take 3 (foldl (++) [] (map sing [1..]))
Interrupted.
```

foldl cu foldr

```
Prelude> let myfoldl h i xs =
                foldr (\x u -> \y -> u (h y x)) id xs i

Prelude> myfoldl (+) 0 [1,2,3]
6

Prelude> let sing = (:[])  -- sing x = [x]

Prelude> take 3 (foldr (++) [] (map sing [1..]))
[1,2,3]

Prelude> take 3 (myfoldl (++) [] (map sing [1..]))
Interrupted.

Prelude> take 3 (foldl (++) [] (map sing [1..]))
Interrupted.
```

Ce se întâmplă?

foldr și foldl



https://en.wikipedia.org/wiki/Fold_(higher-order_function)

- **foldr** poate fi folosită pe liste infinite (în anumite cazuri),
- **foldl** nu poate fi folosită pe liste infinite niciodată.

Programare funcțională

Evaluare leneșă

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

3 noiembrie 2020

Evaluare leneșă. Liste infinite

- Putem folosi funcțile **map** și **filter** pe liste infinite:

```
Prelude> inf = map (+10) [1..] -- inf nu este evaluat
Prelude> take 3 inf
[11,12,13]
```

Limbajul Haskell folosește implicit evaluarea leneșă

- expresiile sunt evaluate numai când este nevoie de valoarea lor
- expresiile nu sunt evaluate total, elementele care nu sunt folosite rămân neevaluate
- o expresie este evaluată o singură dată.

În exemplul de mai sus, este acceptată definiția lui `inf`, fără a fi evaluată. Când expresia **take** 3 `inf` este evaluată, numai primele 3 elemente ale lui `inf` sunt calculate, restul rămânând neevaluate.

Evaluare leneșă: lista numerelor prime

Vă amintiți din primul curs:

```
primes = sieve [2..]
sieve (p:ps) = p : sieve [ x | x <- ps, mod x p /= 0 ]
```

Intuitiv, evaluarea leneșă funcționează astfel:

```
sieve [2..] -->

2 : sieve [ x | x <- [3..], mod x 2 /= 0 ] -->

2 : sieve (3:[ x | x <- [4..], mod x 2 /= 0 ])  -->

2 : 3 : sieve ([ y | y <- [x | x <- [4..], mod x 2 /= 0 ],
                                              mod y 3 /= 0 ])

--> ...
```

foldr și foldl

Definiție

Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare inițială, și o listă, calculați valoarea obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.

Funcția *foldr*

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f i []     = i
foldr f i (x:xs) = f x (foldr f i xs)
```

Funcția *foldl*

```
foldl :: (b -> a-> b) -> b -> [a] -> b
foldl h i []     = i
foldl h i (x:xs) = foldl h (h i  x) xs
```

Funcții de ordin înalt

foldr și foldl

Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare inițială, și o listă, calculați valoare obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.

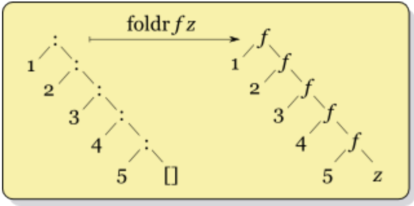
foldr :: (a -> b -> b) -> b -> [a] -> b

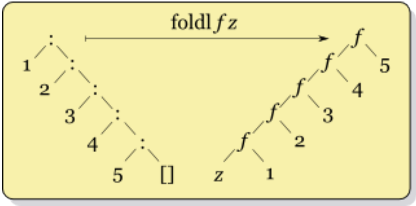
```
foldr op z [a1, a2, a3, ..., an] =
      a1 `op` (a2 `op` (a3 `op` (... (an `op` z) ...)))
```

foldl :: (b -> a -> b) -> b -> [a] -> b

```
foldl op z [a1, a2, a3, ..., an] =
      ( ... (((z `op` a1) `op` a2) `op` a3) ...) `op` an
```

foldr și foldl





https://en.wikipedia.org/wiki/Fold_(higher-order_function)

- **foldr** poate fi folosită pe liste infinite (în anumite cazuri),
- **foldl** nu poate fi folosită pe liste infinite niciodată.

foldr și foldl

- **foldr** poate fi folosită pe liste infinite (în anumite cazuri),
- **foldl** nu poate fi folosită pe liste infinite niciodată.

```
Prelude> foldr (*) 0 [1..]
*** Exception: stack overflow
```

```
Prelude> take 3 $ foldr (\x xs-> (x+1):xs) [] [1..]
[2,3,4]
-- foldr a functionat pe o lista infinita
```

```
Prelude> take 3 $ foldl (\xs x-> (x+1):xs) [] [1..]
-- expresia se calculeaz\u a la infinit
```

Evaluare leneșă. Liste infinite

- Intuitiv, evaluarea leneșă funcționează astfel:

```
foldr (++) [] (map sing [1..]) -->

(++) [1] (foldr (++) [] (map sing [2..]) -->

(++) [1] ((++) [2] (foldr (++) [] (map sing [3..]))) -->

(++) [1] ((++) [2] ((++) [3] (foldr (++) []
                                  (map sing [4..])))) -->
                                      ...
```

- În momentul în care apelăm **take** 3 forțăm evaluarea.

Evaluare leneșă. Liste infinite

- Intuitiv, evaluarea leneșă funcționează astfel:

```
foldr (++) [] (map (:[]) [1..]) -->
(++) [1] (foldr (++) [] (map (:[])[2..]) -->
(++) [1] ((++) [2] (foldr (++) [] (map (:[]) [3..])) -->
```

- În momentul în care apelăm **take** n forțăm evaluarea.
- Deoarece (++) este liniară în primul argument:

```
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

primii n termeni ai expresiei

```
(++) [1] ((++) [2] (foldr (++) [] (map (:[]) [3..])))
```

pot fi determinați fără a calcula toată lista

```
1: ((++) [2] (foldr (++) [] (map (:[]) [3..])) -->
1: 2 : ((++) [3] (foldr (++) [] (map (:[]) [4..])) -->
```

Evaluare leneșă. Liste infinite

- Intuitiv, evaluarea leneșă funcționează astfel:

```
foldl (++) [] (map (:[]) [1..]) -->
foldl (++) [] (1: map (:[]) [2..]) -->
foldl (++) ((++) [1] []) (map (:[]) [2..]) -->
foldl (++) ((++) [1] []) (2: map (:[]) [3..]) -->
foldl (++) ((++) ((++) [1] []) [2]) (map (:[]) [3..]) -->
```

- În cazul lui **foldl** se expresia care calculează rezultatul final trebuie definită complet, ceea ce nu este posibil în cazul listelor infinite.

Optimizarea recursiei folosind evaluarea leneșă

- Să presupunem că vrem să optimizăm generarea șirului Fibonacci

```
f   :: Int -> Integer
f 0 = 0
f 1 = 1
f n = f (n-2) + f (n-1)
```

- o variantă folosind evaluarea leneșă

```
next (a : b : t) = (a + b) : next (b : t)
fibs = 0 : 1 : next fibs
```

Optimizarea recursiei - evaluare leneșă

- o variantă folosind evaluarea leneșă

```
next (a : b : t) = (a + b) : next (b : t)
fibs = 0 : 1 : next fibs
```

Intuitiv, evaluarea leneșă funcționează astfel:

```
fibs --> 0 : 1 : next fibs -->

0 : 1 : next ( 0 : 1 : t) -->          [fibs --> 0 : 1 : t]
0 : 1 : 1 : next (1 : t)  -->          [t --> 1 : next (1 : t)
   ]
0 : 1 : 1 : next (1: 1: t') -->        [t--> 1 : t']
0 : 1 : 1 : 2 : next(1:t')  -->        [t'--> 2 : next (1:t')]
0 : 1 : 1 : 2: next(1:2:t'')  -->      [t'--> 2 : t'']
0 : 1 : 1 : 2 : 3 : next(2:t'')-->     [t''-->3 : next(2:t'')]
```

- Să presupunem că vrem să optimizăm generarea șirului Fibonacci

```
f   :: Int -> Integer
f 0 = 0
f 1 = 1
f n = f (n-2) + f (n-1)
```

  prin reținerea și accesarea directă a valorilor anterior calculate (*memoizare*).
- Haskell este un limbaj *stateless*, nu avem posibilitatea de a reține valorile într-un vector, așa cum am face într-un limbaj imperativ.

  Cum procedăm?

Optimizarea recursiei

- Să presupunem că vrem să optimizăm generarea șirului Fibonacci

```
f :: Int -> Integer
f 0 = 0
f 1 = 1
f n = f (n-2) + f (n-1)
```

  prin reținerea valorilor anterioare.

- În Haskell putem reține valorile generate de o funcție într-o listă folosind funcția **map**

```
genf :: Int -> Integer
genf n = (map f [1..]) !! n
```

  Obervați că:

  - folosim *evaluarea leneșă* pentru a construi lista *tuturor* numerelor
  - accesăm elementul n din lista

  Nu am rezolvat problema optimizării,
  dar am găsit o modalitate de a construi lista valorilor.

Optimizarea recursiei

- Deoarece știm cum să construim lista de valori, putem elimina apelul recursiv cu accesarea elementelor listei:

```
f :: Int -> Integer
f 0 = 0
f 1 = 1
f n = genf  (n-2) + genf  (n-1)  -- am inlocuit
                                  -- apelul recursiv

genf :: Int -> Integer
genf n = (map f [1..]) !! n
```

- Credeți că am rezolvat problema? Nu, deoarece listele care calculează rezultatele în genf (n−2) și genf (n−1) sunt diferite. Fiecare apel al lui f creaza liste noi, de fapt complexitatea crește.

  Trebuie să găsim o soluție în care să folosim o singură listă.

Optimizarea recursiei: memoizare

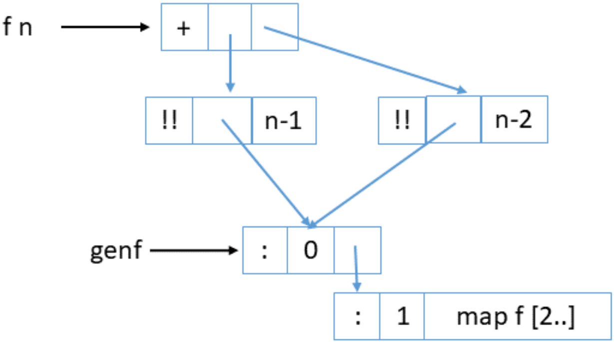
O soluția corectă:

```
f :: Int -> Integer
f 0 = 0
f 1 = 1
f n = (genf !! (n-2)) + (genf !! (n-1))


genf = map f [0..]
```

```
*Main> f 200
280571172992510140037611932413038677189525
(0.01 secs, 206,448 bytes)
```

Optimizarea recursiei: memoizare

In Haskell expresiile sunt reprezentate sub forma unor grafuri:



Elementele lui genf sunt evaluate o singură dată!

Programare funcțională

Clase de tipuri

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Exemplu: test de apartenență

Să scriem funcția **elem** care testează dacă un element aparține unei liste.

- definiția folosind descrieri de liste

```
elem x ys      = or [ x == y | y <- ys ]
```

- definiția folosind recursivitate

```
elem x []      = False
elem x (y:ys)  = x == y || elem x ys
```

- definiția folosind funcții de nivel înalt

```
elem x ys      = foldr (||) False (map (x ==) ys)
```

Funcția elem este polimorfică

```
*Main> elem 1 [2,3,4]
False

*Main> elem 'o' "word"
True

*Main> elem (1,'o') [(0,'w'),(1,'o'),(2,'r'),(3,'d')]
True

*Main> elem "word" ["list","of","word"]
True
```

Care este tipul funcției **elem**?

Funcția elem este polimorfică

```
*Main> elem 1 [2,3,4]
False

*Main> elem 'o' "word"
True

*Main> elem (1,'o') [(0,'w'),(1,'o'),(2,'r'),(3,'d')]
True

*Main> elem "word" ["list","of","word"]
True
```

Care este tipul funcției **elem**?

Funcția **elem** este polimorfică.

Definiția funcției este parametrică în tipul de date.

Funcția elem este polimorfică

Dar nu pentru orice tip

- Totuși definiția nu funcționează pentru orice tip!

```
*Main> elem (+ 2) [(+ 2), sqrt]
No instance for (Eq (Double -> Double)) arising from a use of ‘elem‘
```

Ce se întâmplă?

```
> :t elem_
elem_ :: Eq a => a -> [a] -> Bool
```

În definiția

```
elem x ys      = or [ x == y | y <- ys ]
```

folosim relația de egalitate == care nu este definită pentru orice tip:

```
Prelude> sqrt == sqrt
No instance for (Eq (Double -> Double)) ...
Prelude> ("ab",1) == ("ab",2)
False
```

Clase de tipuri

- O clasă de tipuri este determinată de o mulțime de funcții (este o interfață).

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- minimum definition: (==)
  x /= y = not (x == y)
  -- ^^^ putem avea definitii implicite
```

- Tipurile care aparțin clasei sunt instanțe ale clasei.

```haskell
instance Eq Bool where
  False == False = True
  False == True  = False
  True  == False = False
  True  == True  = True
```

Clasa de tipuri. Constrângeri de tip

- În signatura funcției **elem** trebuie să precizăm ca tipul a este în clasa **Eq**

  ```
  elem :: Eq a => a -> [a] -> Bool
  ```

  Eq a se numește constrângere de tip. **=>** separă constrângerile de tip de restul signaturii.

- În exemplul de mai sus am considerat că **elem** este definită pe liste, dar în realitate funcția este mai complexă

  ```
  Prelude> :t elem
  elem :: (Eq a, Foldable t) => a -> t a -> Bool
  ```

  În această definiție Foldable este o altă clasă de tipuri, iar t este un parametru care ține locul unui *constructor de tip*!

Sistemul tipurilor in Haskell este complex!

Instanțe ale lui Eq

```haskell
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int        where
  (==) = eqInt    -- built-in

instance    Eq Char     where
  x == y               = ord x == ord y

instance (Eq a, Eq b) => Eq (a,b) where
  (u,v) == (x,y)       = (u == x) && (v == y)

instance Eq a => Eq [a] where
  [] == []             = True
  [] == y:ys           = False
  x:xs == []           = False
  x:xs == y:ys         = (x == y) && (xs == ys)
```

Eq, Ord

- Clasele pot fi extinse

```
class   (Eq a) => Ord a   where
  (<)   ::   a -> a ->   Bool
  (<=)  ::   a -> a ->   Bool
  (>)   ::   a -> a ->   Bool
  (>=)  ::   a -> a ->   Bool
  -- minimum   definition: (<=)
  x < y    =   x <= y && x /= y
  x > y    =   y < x
  x >= y   =   y <= x
```

Clasa **Ord** este clasa tipurilor de date înzestrate cu o relație de ordine.

În definiția clasei **Ord** s-a impus o constrângere de tip. Astfel, orice instanță a clasei **Ord** trebuie să fie instanță a clasei **Eq**.

Instanțe ale lui **Ord**

```
instance  Ord Bool  where
    False <= False = True
    False <= True  = True
    True  <= False = False
    True  <= True  = True

instance (Ord a, Ord b) => Ord (a,b) where
  (x,y) <= (x',y') = x < x' || (x == x' && y <= y')
  -- ordinea lexicografica

instance  Ord a => Ord [a]  where
    []     <= ys     = True
    (x:xs) <= []     = False
    (x:xs) <= (y:ys) = x < y || (x == y && xs <= ys)
```

Definirea claselor

Să presupunem că vrem să definim o clasă de tipuri pentru datele care pot fi afișate. O astfel de clasă trebuie să conțină o metodă care să indice modul de afișare:

```
class  Visible a  where
    toString :: a -> String
```

Putem face instanțieri astfel:

```
instance  Visible Char where
    toString c = [c]
```

Clasele **Eq**, **Ord** sunt predefinite. Clasa Visible este definită de noi, dar există o clasă predefinită care are același rol: clasa **Show**

Show

```haskell
  class  Show a  where
    show :: a -> String      -- analogul lui "toString"

instance  Show Bool  where
    show False       = "False"
    show True        = "True"

instance (Show a, Show b) => Show (a,b) where
  show (x,y) = "(" ++ show x ++ "," ++ show y ++ ")"

  instance Show a => Show [a] where
    show []     = "[]"
    show (x:xs) = "[" ++ showSep x xs ++ "]"
      where
        showSep x []     = show x
        showSep x (y:ys) = show x ++ "," ++ showSep y ys
```

Clase de tipuri pentru numere

```
class (Eq a, Show a) => Num a where
  (+),(-),(*)     :: a -> a -> a
  negate          :: a -> a
  fromInteger     :: Integer -> a
  -- minimum definition: (+),(-),(*),fromInteger
  negate x       =   fromInteger 0 - x

class (Num a) => Fractional a where
  (/)             :: a -> a -> a
  recip           :: a -> a
  fromRational  :: Rational -> a
  -- minimum definition: (/), fromRational
  recip x        =   1/x

class (Num a, Ord a) => Real a where
  toRational      :: a -> Rational

class (Real a, Enum a) => Integral a where
  div, mod        :: a -> a -> a
  toInteger       :: a -> Integer
```

Clasa de tipuri Enum

```
class Enum a where
  toEnum          ::   Int -> a
  fromEnum        ::   a -> Int
  succ, pred      ::   a -> a
  enumFrom        ::   a -> [a]                 --   [x..]
  enumFromTo      ::   a -> a -> [a]            --   [x..y]
  enumFromThen    ::   a -> a -> [a]            --   [x,y..]
  enumFromThenTo  ::   a -> a -> a -> [a]       --   [x,y..z]
 -- minimum definition: toEnum, fromEnum
 succ x              = toEnum (fromEnum x + 1)
 pred x              = toEnum (fromEnum x - 1)
 enumFrom x
   = map toEnum [fromEnum x ..]
 enumFromTo x y
   = map toEnum [fromEnum x .. fromEnum y]
 enumFromThen x y
   = map toEnum [fromEnum x, fromEnum y ..]
 enumFromThenTo x y z
   = map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

Int ca instanță a lui Enum

```
instance Enum Int where
  toEnum x            = x
  fromEnum x          = x
  succ x              = x+1
  pred x              = x-1
  enumFrom x          = iterate (+1) x
  enumFromTo x y      = takeWhile (<= y) (iterate (+1) x)
  enumFromThen x y    = iterate (+(y-x)) x
  enumFromThenTo x y z
      = takeWhile (<= z) (iterate (+(y-x)) x)

iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

takeWhile :: (a -> Bool) -> [a]   ->   [a]
takeWhile p []                    =    []
takeWhile p (x:xs) | p x          =    x : takeWhile p xs
                   | otherwise    =    []
```

Anotimpuri

```
data Season = Winter | Spring | Summer | Fall

next   :: Season -> Season
next   Winter = Spring
next   Spring = Summer
next   Summer = Fall
next   Fall   = Winter

warm   :: Season -> Bool
warm   Winter = False
warm   Spring = True
warm   Summer = True
warm   Fall   = True
```

Anotimpuri — Instanțiere manuala pentru Eq, Ord, Show

```haskell
instance    Eq    Seasons    where
  Winter    ==    Winter    = True
  Spring    ==    Spring    = True
  Summer    ==    Summer    = True
  Fall      ==    Fall      = True
  _         ==    _         = False
instance    Ord  Seasons  where
  Spring    <=  Winter  = False
  Summer    <=  Winter  = False
  Summer    <=  Spring  = False
  Fall      <=  Winter  = False
  Fall      <=  Spring  = False
  Fall      <=  Summer  = False
  _         <=  _         = True
instance  Show  Seasons  where
  show  Winter  =  "Winter"
  show  Spring  =  "Spring"
  show  Summer  =  "Summer"
  show  Fall    =  "Fall"
```

Instanță Enum pentru anotimpuri

```
instance Enum Seasons where

  fromEnum    Winter  =  0
  fromEnum    Spring  =  1
  fromEnum    Summer  =  2
  fromEnum    Fall    =  3

  toEnum  0  =  Winter
  toEnum  1  =  Spring
  toEnum  2  =  Summer
  toEnum  3  =  Fall
```

Anotimpuri folosind derivare automată

```
data Season = Winter | Spring | Summer | Fall
              deriving (Eq, Ord, Show, Enum)

next :: Season -> Season
next x = toEnum ((fromEnum x + 1) `mod` 4)

warm :: Season -> Bool
warm x = x `elem` [Spring .. Fall]
```

Toate funcțiile pentru clasele derivate sunt generate automat!

Programare funcțională

Date structurate.

Ioana Leuștean
Traian Florin Șerbănuță

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

Tipuri sumă

- În Haskell tipul **Bool** este definit astfel:

```
data Bool = False | True
```

  Bool este constructor de tip
  False și **True** sunt constructori de date

- În mod similar putem defini

```
data Season = Spring | Summer
            | Autumn | Winter
```

  Season este constructor de tip
  Spring, Summer, Autumn și Winter sunt constructori de date

Bool și Season sunt tipuri de date sumă, adică sunt definite prin enumerarea alternativelor.

Tipuri sumă

```
data Bool = False | True
```

- Operațiile se definesc prin "pattern matching":

```
not :: Bool -> Bool
not False  =  True
not True   =  False

(&&), (||) :: Bool -> Bool -> Bool
False && q   =  False
True  && q   =  q
False || q   =  q
True  || q   =  True
```

Tip sumă: anotimpuri

```haskell
data Season = Spring | Summer
            | Autumn | Winter

  succesor  Spring = Summer
  succesor  Summer = Autumn
  succesor  Autumn = Winter
  succesor  Winter = Spring

  showSeason Spring = "Primavara"
  showSeason Summer = "Vara"
  showSeason Autumn = "Toamna"
  showSeason Winter = "Iarna"
```

Tipuri produs

- Să definim un tip de date care să aibă ca valori "punctele" cu două coordonate de tipuri oarecare:

```
data Point a b = Pt a b
```

  Point este constructor de tip
  Pt este constructor de date

- Pentru a accesa componentele, definim proiecțiile:

```
pr1 : Point a b -> a
pr1 (Pt x _) = x
pr2 : Point a b -> b
pr2 (Pt _ y) = y
```

Point este un tip de date produs, definit prin combinarea tipurilor a și b.

Tipuri produs

```
data Point a b = Pt a b

Prelude> :t (Pt 1 "c")
(Pt 1 "c") :: Num a => Point a [Char]

Prelude> :t Pt
Pt :: a -> b -> Point a b
-- constructorul de date este operatie

Prelude> :t (Pt 1)
(Pt 1) :: Num a => b -> Point a b
```

- Se pot defini operații:

```
pointFlip :: Point a b -> Point b a
pointFlip (Pt x y) = Pt y x
```

Tipuri de date definite recursiv

- Declarația listelor ca tip de date algebric

```
data  List a  = Nil
              | Cons a (List a)
```

- Se pot defini operații

```
append :: List a -> List a -> List a
append Nil ys          = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Tipuri de date algebrice

Tipurile de date algebrice se definesc folosind "operațiile" sumă și produs.

Forma generală

$$
\begin{array}{lcl}
\mathit{data}\ \mathit{Typename} & = & \mathit{Cons}_1\ \ t_{11}\ldots t_{1k_1} \\
 & & \mid \mathit{Cons}_2\ \ t_{21}\ldots t_{2k_2} \\
 & & \mid \ldots \\
 & & \mid \mathit{Cons}_n\ \ t_{n1}\ldots t_{nk_n}
\end{array}
$$

unde $k_1, \ldots, k_n \geq 0$

- Se pot folosi tipuri sumă și tipuri produs.
- Se pot defini tipuri parametrizate.
- Se pot folosi definiții recursive.

Tipuri de date algebrice

Forma generală

$$
\begin{aligned}
data\ \ Typename \quad = \quad & \ Cons_1 \ \ t_{11} \ldots t_{1k_1} \\
& \mid Cons_2 \ \ t_{21} \ldots t_{2k_2} \\
& \mid \ldots \\
& \mid Cons_n \ \ t_{n1} \ldots t_{nk_n}
\end{aligned}
$$

unde $k_1, \ldots, k_n \geq 0$

Atenție! Alternativele trebuie să conțină constructori.

data StrInt = **String** | **Int** este greșit

data StrInt = VS **String** | VI **Int** este corect

[VI 1, VS "abc", VI 34, VI 0, VS "xyz"] :: [StrInt]

Tipuri de date algebrice - exemple

```
data Bool = False | True

data Season = Winter | Spring | Summer | Fall

data Shape = Circle Float | Rectangle Float Float

data Maybe a = Nothing | Just a

data Pair a b = Pair a b
    -- constructorul de tip si cel de date pot sa coincida

data Nat = Zero | Succ Nat

data Exp = Lit Int | Add Exp Exp | Mul Exp Exp

data List a = Nil | Cons a (List a)

data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
```

Constructori simboluri

```haskell
data List a = Nil
            | Cons a (List a)
```

Declarație ca tip de date algebric cu simboluri

```haskell
data List a = Nil
            | a ::: List a
     deriving (Show)

infixr 5 :::
```

Liste și tupluri

- Liste

```haskell
data [a] = [] | a : [a]
```

Constructorii listelor sunt [] și : unde

```haskell
[] :: [a]
(:) :: a -> [a] -> [a]
```

- Tupluri

```haskell
data (a,b) = (a,b)
data (a,b,c) = (a,b,c)
  ...                   ...
```

Nu exisă o declarație generică pentru tupluri, fiecare declarație de mai sus definește tuplul de lungimea corespunzătoare, iar constructorii pentru fiecare tip în parte sunt:

```haskell
(,) :: a -> b -> (a,b)
(,,) :: a -> b -> c  -> (a,b,c)
  ...
```

Utilizarea **type**

Cu **type** se pot redenumi tipuri deja existente.

```
type FirstName  = String
type LastName   = String
type Age        = Int
type Height     = Float
type Phone = String

data Person = Person FirstName LastName Age Height Phone
```

Exemplu - date personale. Proiecții

```
data Person = Person FirstName LastName Age Height Phone

    firstName :: Person -> String
    firstName (Person firstname _ _ _ _ _) = firstname

    lastName :: Person -> String
    lastName (Person _ lastname _ _ _ _) = lastname

    age :: Person -> Int
    age (Person _ _ age _ _ _) = age

    height :: Person -> Float
    height (Person _ _ _ height _ _) = height

    phoneNumber :: Person -> String
    phoneNumber (Person _ _ _ _ number _) = number
```

Exemplu - date personale. Utilizare

```
Main*> let ionel = Person "Ion" "Ionescu" 20 175.2 "
    0712334567"
```

```
Main*> firstName ionel
"Ion"
```

```
Main*> height ionel
175.2
```

```
Main*> phoneNumber ionel
"0712334567"
```

Date personale ca înregistrări

- Putem folosi atât forma algebrică cât și cea de înregistrare

```
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"

gigel = Person { firstName = "Gheorghe"
               , lastName="Georgescu"
               , age = 30, height = 192.3
               , phoneNumber = "0798765432"
               }
```

- Putem folosi și pattern-matching
- Proiecțiile sunt definite automat; sintaxă specializată pentru actualizări

```
nextYear :: Person -> Person
nextYear person = person { age = age person + 1 }
```

Date personale ca înregistrări

```haskell
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     }
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"

nextYear person = person { age = age person + 1 }
```

```
*Main> nextYear ionel
 No instance for (Show Person) arising from a use of ‘print’
```

Deși toate definițiile sunt corecte, o valoare de tip Person nu poate fi afișată deoarece nu este instanță a clasei **Show**.

Derivare automata pentru tipuri algebrice

Am definit tipuri de date noi:

```haskell
data Season = Spring | Summer | Autumn | Winter
              deriving (Eq, Ord, Show)

data Point a b = Pt a b
                 deriving (Eq, Ord, Show)
```

Cum putem să le facem instanțe ale claselor **Eq**, **Ord**, **Show**?

Putem să le facem explicit sau să folosim derivarea automată.

Atenție!
Derivarea automată poate fi folosită numai pentru unele clase predefinite.

Derivare automata vs Instanțiere explicită

- O clasă de tipuri este determinată de o mulțime de funcții.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- minimum definition: (==)
  x /= y = not (x == y)
```

- Tipurile care aparțin clasei sunt instanțe ale clasei.
- Instanțierea prin derivare automată:

```
data Point a b = Pt a b
                 deriving Eq
```

- Instanțiere explicită:

```
instance Eq a => Eq (Point a b) where
     (==) (Pt x1 y1) (Pt x2 y2) = (x == x1)
```

Derivare automata pentru tipuri algebrice

```
data Point a b = Pt a b
                 deriving (Eq, Ord, Show)
```

Egalitatea, relația de ordine și modalitatea de afișare sunt definite implicit dacă este posibil:

```
*Main> Pt 2 3 < Pt 5 6
True

*Main> Pt 2 "b" < Pt 2 "a"
False

*Main Data.Char> Pt (+2) 3 < Pt (+5) 6
```

No instance for (Ord (Integer -> Integer))arising from a use of ‘<’

Instanțiere explicită - exemplu

```
data Season = Spring | Summer | Autumn | Winter

Instance Eq Season where
  Spring == Spring = True
  Summer == Summer = True
  Autumn == Autumn = True
  Winter == Winter = True
  _      ==    _   = False


Instance Show Season where
  show Spring = "Primavara"
  show Summer = "Vara"
  show Autumn = "Toamna"
  show Winter = "Iarna"
```

Exemplu: numerele naturale (Peano)

Cum definim numerele naturale?

Declarație ca tip de date algebric folosind șabloane

```
data Nat = Zero | Succ Nat
```

- Putem să definim operații

```
(^^^) :: Float -> Nat -> Float
x ^^^ Zero     = 1.0
x ^^^ (Succ n) = x * x ^^^ n
```

Comparați cu versiunea folosind notația predefinită

```
(^^) :: Float -> Int -> Float
x ^^ 0 = 1.0
x ^^ n = x * (x ^^ (n-1))
```

Exemplu: adunare și înmulțire pe Nat

Definiție pe tipul de date algebric

```haskell
(+++) :: Nat -> Nat -> Nat
m +++ Zero     = m
m +++ (Succ n) = Succ (m +++ n)

(***) :: Nat -> Nat -> Nat
m *** Zero     = Zero
m *** (Succ n) = (m *** n) +++ m
```

Comparați cu versiunea folosind notația predefinită

```haskell
(+) :: Int -> Int -> Int
m + 0 = m
m + n = (m + (n-1)) + 1

(*) :: Int -> Int -> Int
m * 0 = 0
m * n = (m * (n-1)) + m
```

Exemplu: liste

```
data  List a  = Nil
              | a ::: List a
          deriving (Show)

infixr 5 :::
```

- Putem defini operații:

```
(+++) :: List a -> List a -> List a
infixr 5 +++
Nil +++ ys          = ys
(x ::: xs) +++ ys = x ::: (xs +++ ys)
```

Comparați cu versiunea folosind notația predefinită

```
(++) :: [a] -> [a] -> [a]
[] ++ ys       = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Constructori simboluri

Definirea egalității și a reprezentării

```haskell
eqList :: Eq a => List a -> List a -> Bool
eqList Nil Nil  = True
eqList (x ::: xs)  (y:::ys) = x == y && eqList xs ys
eqList _           _                  = False

instance (Eq a) => Eq (List a) where
   (==) = eqList


showList :: Show a => List a -> String
showList Nil = "Nil"
showList (x ::: xs) = show x ++ " ::: " ++ showList xs

instance (Show a) => Show (List a) where
   show = showList
```

Tipul Maybe (opțiune)

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

Argumente opționale

```haskell
power :: Maybe Int -> Int -> Int
power Nothing n   = 2 ^ n
power (Just m) n = m ^ n
```

Rezultate opționale

```haskell
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n `div` m)
```

Maybe - folosirea unui rezultat opțional

```haskell
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n `div` m)

-- utilizare gresita
wrong :: Int -> Int -> Int
wrong n m = divide n m + 3

-- utlizare corecta
right :: Int -> Int -> Int
right n m = case divide n m of
              Nothing -> 3
              Just r -> r + 3
```

Either A B (A sau B)

```
data Either a b = Left a | Right b

mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Definiți o funcție care calculează suma elementelor întregi.

```
addints     :: [Either Int String] -> Int
addints     []                = 0
addints     (Left n : xs)     = n + addints xs
addints     (Right s : xs) = addints xs


addints' :: [Either Int String] -> Int
addints' xs = sum [n | Left n <- xs]
```

A sau B

```haskell
data Either a b = Left a | Right b
```

```haskell
mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Definiți o funcție care întoarce concatenarea elementelor de tip **String**.

```haskell
addstrs    :: [Either Int String] -> String
addstrs    []                = ""
addstrs    (Left n : xs)     = addstrs xs
addstrs    (Right s : xs) = s ++ addstrs xs
```

```haskell
addstrs' :: [Either Int String] -> String
addstrs' xs = concat [s | Right s <- xs]
```

Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Testare QuickCheck - Exemplu

K. Claessen, J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". Proceedings of the ICFP, ACM SIGPLAN, 2000.

```
import Test.QuickCheck

myreverse :: [Int] -> [Int]
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]

prdef :: [Int] -> Bool
prdef xs = (myreverse xs == reverse xs)

*Main> quickCheck prdef
+++ OK, passed 100 tests.
```

Testare QuickCheck - Exemplu

```
import Test.QuickCheck


myreverseW :: [Int] -> [Int]
myreverseW [] = []
myreverseW (x:xs) = x:(myreverse1 xs)

prdefW :: [Int] -> Bool
prdefW xs = (myreverseW xs == reverse xs)

*Main> quickCheck prW
*** Failed! Falsified (after 4 tests and 5 shrinks):
[0,1]
```

Clasa tipurilor "mici"

http://www.cse.chalmers.se/edu/year/2018/course/TDA452/lectures/OverloadingAndTypeClasses.html

- Definiți clasa tipurilor de date cu un număr "mic" de valori.

```
class MySmall a where
  smallValues :: [a]

instance MySmall Bool where
   smallValues = [True, False]

data Season = Spring | Summer | Autumn | Winter
              deriving Show
instance MySmall Season  where
   smallValues = [Spring, Summer, Autumn, Winter]

> smallValues :: [Season] -- trebuie sa precizam tipul
[Spring,Summer,Autumn,Winter]
```

Clasa tipurilor "mici"

```haskell
class MySmall a where
  smallValues :: [a]


instance MySmall Int where
   smallValues = [0,12,3,45,91,100]

instance (MySmall s) => MySmall (a -> s) where
   smallValues = [const v | v <- smallValues]
   -- const v _ = v

> sv = smallValues :: [String -> Season]
> length sv
4
> head sv $ "lalalal"
Spring
> sv !! 2 $ "blabla"
Autumn
```

Clasa tipurilor "mici" - testare

- Definiți o clasa care conține o funcție asemănătoare cu quickCheck care testează dacă o proprietate este adevărată pentru toate valorile unui tip "mic".

```
class  MySmallCheck a where
  smallValues :: [a]
  smallCheck :: (a -> Bool) -> Bool
  smallCheck prop = and [ prop x | x <- smallValues ]
  -- minimal definition: smallValues
```

Clasa tipurilor "mici" - testare

```haskell
class  MySmallCheck a where
   smallValues :: [a]
   smallCheck :: (a -> Bool) -> Bool
   smallCheck prop = and [ prop x | x <- smallValues ]

instance MySmallCheck Int where
   smallValues = [0,12,3,45,91,100]


propInt :: Int -> Bool
propInt x = x < 90
propInt1 :: Int -> Bool
propInt1 x = x < 101

> smallCheck propInt
False
> smallCheck propInt1
True
```

Clasa tipurilor "mici" - testare

- Putem defini smallCheck astfel încât să precizeze un contraexemplu?

```
class MySmallCheck a where
   smallValues :: [a]
   smallCheck :: (a -> Bool) -> Bool
   smallCheck prop = sc smallValues
                          where
                            sc [] = True
                            sc (x:xs) = if (prop x)
                                        then (sc xs)
                                        else error "..."
instance MySmallCheck Int where
   smallValues = [0,12,3,45,91,100]

propInt :: Int -> Bool
propInt x = x < 90

> smallCheck propInt
*** Exception: False! Counterexample:91
```

PRNG

Ce facem cand avem tipuri cu un numar mare de valori (asa cum este **Int**)?
Trebuie să generăm valori pseudo-aleatoare.

PRNG

Un *Pseudo random number generator* este un algoritm care produce o secvența de numere aleatoare, având ca punct de plecare o valoare inițială (*seed*).

Exemplu:

$$\begin{aligned} \text{Linear Congruence Generator:} \quad X_{i+1} &= aX_i + c \ (mod\ m) \\ seed &= X_0 \end{aligned}$$

Exemplu: numere aleatoare între 0 și 10

- Generator de numere aleatoare

```
rval i = (7 * i + 3 ) `mod` 11  -- valori intre 0 si 10
```

```
> rval 0 -- samanta este 0
3        -- valorea aleatoare generata
```

- Generăm o secvență de numere aleatoare

```
genRandSeq 0 _ = []
genRandSeq n s = let news = rval s
                 in news : (genRandSeq (n-1) news)

-- n este numarul de valori care vor fi generate
-- s este samanta
```

Exemplu: numere aleatoare între 0 și 10

- Generăm o secvență de numere aleatoare

```
rval i = (7 * i + 3 ) `mod` 11  -- valori intre 0 si 10

genRandSeq 0 _ = []
genRandSeq n s = let news = rval seed
                 in news : (genRandSeq (n-1) news)
```

```
> genRandSeq 10 0
[3,2,6,1,10,7,8,4,9,0]

> genRandSeq 20 0
[3,2,6,1,10,7,8,4,9,0,3,2,6,1,10,7,8,4,9,0]
```

Secvența aleatoare este predictibilă. Cum îmbunătățim algoritmul?

Exemplu: numere aleatoare între 0 și 10

- Folosim generatoare dfierite pentru valori si semințe

```
rval i = (7 * i + 3 ) `mod` 11  -- valori intre 0 si 10
rseed i = (7 * i + 3 ) `mod` 101

genRandSeq 0 _ = []
genRandSeq n s = let
                    val  = rval s
                    news = rseed s
                 in  (val : (genRandSeq (n-1) news) )

> genRandSeq 10 0
[3,2,6,9,10,0,0,8,8,3]
> genRandSeq 20 0
[3,2,6,9,10,0,0,8,8,3,7,2,3,9,5,4,6,6,3,10]
> genRandSeq 30 0
[3,2,6,9,10,0,0,8,8,3,7,2,3,9,5,4,6,6,3,10,9,4,3,6,1,3,4,5,
9,2]
```

PRNG: valorile și semințele sunt diferite

```
type Seed = Int
type RValue = Int

myrand :: Seed -> (RValue, Seed)
myrand i = (rval i, rseed i)

genRandSeq 0 _ = []
genRandSeq n  s = let (val,news) = myrand s
                  in  (val : (genRandSeq (n-1) news) )
```

Generarea numerelor aleatoare în Haskell

http://hackage.haskell.org/package/random-1.1/docs/System-Random.html

```
class RandomGen g where
   next :: g -> (Int,g)
 -- observati asemanarea cu myrand :: Seed ->(RValue, Seed)
   ...

data StdGen
instance RandomGen StdGen where ...

mkStdGen :: Int -> StdGen

---  pt tipuri oarecare
class Random a where
   random :: RandomGen g => g -> (a, g)
   randoms :: RandomGen g => g -> [a]
   randomRs :: RandomGen g => (a, a) -> g -> [a]
   ....
```

Generarea numerelor aleatoare în Haskell

http://hackage.haskell.org/package/random-1.1/docs/System-Random.html

```
System.Random> genInt = fst $ random (mkStdGen 1000) :: Int

System.Random> genInt
1611434616111168504
System.Random> genInt
1611434616111168504

System.Random> genInts = randoms (mkStdGen 500) :: [Int]

System.Random> take 10 genInts
[−8476283234809671955,5851875716463766781,−1174332976046471371

−6005536961401157228,1127019136727650924,−5427348788055872176,

−3587680396420832273,−1231390686875326875,4168674226095003295,

−6936465015900757066]
```

Generarea caracterelor aleatoare în Haskell

http://hackage.haskell.org/package/random-1.1/docs/System-Random.html

```
System.Random> genChar = fst$randomR ('a','z') (mkStdGen
    500)::Char
System.Random> genChar
'x'
System.Random> genChar
'x'
System.Random> genChars = randomRs ('a','z')(mkStdGen 500)::
    [Char]
System.Random> take 10 genChars
"xofmefswxj"
System.Random> take 50 genChars
"xofmefswxjxyhuuuditkpdrrqrhbdsfyyyhtfutowrxlnszfct"
```

Testare QuickCheck - Exemplu

```
import Test.QuickCheck

myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]


prdef xs = (myreverse xs == reverse xs)
wrongpr xs = myreverse xs == xs

> quickCheck prdef
+++ OK, passed 100 tests.

> quickCheck wrongpr
+++ OK, passed 100 tests.
```

Ce se întâmplă?

Testare QuickCheck - Exemplu

```
import Test.QuickCheck
myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]

> verboseCheck wrongpr
...
Passed:
[(),(),(),(),(),(),(),(),()]
Passed:
[(),(),(),()]
Passed:
[(),(),(),()]
Passed:
[(),(),(),(),(),(),(),(),(),(),(),()]
Passed:
[(),(),(),(),(),(),()]
...
```

Testare QuickCheck - Exemplu

Trebuie să precizăm tipul datelor testate!

```
import Test.QuickCheck
myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]

prdef :: [Int] -> Bool  -- precizam tipul
prdef xs = (myreverse xs == reverse xs)
wrongpr :: [Int] -> Bool  -- precizam tipul
wrongpr xs = myreverse xs == xs

> quickCheck prdef
+++ OK, passed 100 tests.

> quickCheck wrongpr
*** Failed! Falsified (after 4 tests and 3 shrinks):
[1,0]
```

Testare QuickCheck - ADT

```haskell
data Season = Spring | Summer | Autumn | Winter
                deriving (Show, Eq)

prdef1 :: [Season] -> Bool
prdef1 xs = (myreverse xs == reverse xs)

wrongpr1 :: [Season] -> Bool
wrongpr1 xs = myreverse xs == xs
```

```
> quickCheck prdef1
```

error:

No instance for (Arbitrary Season)

Testare QuickCheck

- Generarea testelor aleatoare depinde de tipul de date.
- Tipurile de date care pot fi testate cu QuickCheck trebuie să fie instanțe ale clasei Arbitrary :

```
class Arbitrary a where
  arbitrary :: Gen a
```

- Gen a este un "wrapper" pentru un alt generator

```
newtype Gen a = MkGen{ unGen :: QCGen -> Int -> a}
```

unde QCGen poate fi definit folosind, de exemplu, **StdGen**

```
newtype QCGen = QCGen StdGen
```

http://hackage.haskell.org/package/QuickCheck-2.13.2/docs/src/Test.QuickCheck.Random.html

http://hackage.haskell.org/package/QuickCheck-2.13.2/docs/Test-QuickCheck.html

Testare QuickCheck

- Tipurile de date care pot fi testate cu QuickCheck trebuie să fie instanțe ale clasei Arbitrary :

```
class Arbitrary a where
  arbitrary :: Gen a
```

- Gen a poate fi tratat ca un tip abstract, datele de tip Gen a pot fi definite cu ajutorul combinatorilor:

```
choose :: Random a => (a, a) -> Gen a
oneof :: [Gen a] -> Gen a
elements :: [a] -> Gen a
....
```

Testare QuickCheck - ADT

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Show, Eq)

instance Arbitrary Season where
    arbitrary = elements [Spring, Summer, Autumn, Winter]

prdef1 :: [Season] -> Bool
prdef1 xs = (myreverse xs == reverse xs)
wrongpr1 :: [Season] -> Bool
wrongpr1 xs = myreverse xs == xs

> quickCheck prdef1
+++ OK, passed 100 tests.

> quickCheck wrongpr1
*** Failed! Falsified (after 3 tests):
[Winter,Summer]
```

Testare QuickCheck - ADT

```
newtype MyInt = My Int
                  deriving (Show, Eq)

instance Arbitrary MyInt where
    arbitrary =   elements (map My listInt)
                  where listInt = take 500000 (randoms (
                      mkStdGen 0)) :: [Int]

prdef1 :: [Season] -> Bool
prdef1 xs = (myreverse xs == reverse xs)
wrongpr1 :: [Season] -> Bool
wrongpr1 xs = myreverse xs == xs

> quickCheck prdef1
+++ OK, passed 100 tests.
> quickCheck wrongpr1
*** Failed! Falsified (after 3 tests):
[Winter,Summer]
```

Testare QuickCheck - ADT

```
newtype MyInt = My Int
                deriving (Show, Eq)
```

Cum definim instanța lui Arbitrary? O variantă ar fi tot folosirea operației elements:

```
instance Arbitrary MyInt where
   arbitrary = elements (map My listInt)
               where listInt = [-1000000, 1000000]
```

Putem genera lista de întregi:

```
import System.Random
randoms :: RandomGen g => g -> [a]
        -- instance RandomGen StdGen

instance Arbitrary MyInt where
   arbitrary = elements (map My listInt)
            where
          listInt = take 500000(randoms(mkStdGen 0))::[Int]
```

Testare QuickCheck - ADT

```
import System.Random
newtype MyInt = My Int
                deriving (Show, Eq)

instance Arbitrary MyInt where
    arbitrary =  elements (map My listInt)
             where
            listInt = take 500000(randoms(mkStdGen 0))::[Int]

wrongpr2 :: [MyInt] -> Bool
wrongpr2 xs = myreverse xs == xs

> quickCheck wrongpr2
*** Failed! Falsified (after 5 tests and 1 shrink):
[My 4948157297514287243,My (-2390719447972180436)]
```

Testare QuickCheck - ADT

```haskell
newtype MyInt = My Int
                deriving (Show, Eq)
```

Putem defini instanța lui Arbitrary folosind direct definiția datelor de tip *Gen a*

```haskell
newtype Gen a = MkGen{ unGen :: QCGen -> Int -> a}
```

Știind că **Int** este instanță a lui Arbitrary, să definim o instanță pentru MyInt.

```haskell
instance Arbitrary MyInt where
   arbitrary = MkGen (\s i -> let x = f s i in (My x))
                where
                  f = (unGen (arbitrary :: Gen Int))
```

Testable

```
Prelude> import Test.QuickCheck
Prelude Test.QuickCheck> :t quickCheck
quickCheck :: Testable prop => prop -> IO ()
```

- Pentru a putea fi testată o proprietate trebuie să aparțină unei instanțe a clasei Testable
- Instanță trivială

```
instance Testable Bool where
   ...

Prelude Test.QuickCheck> quickCheck (1 + 2 == 3)
+++ OK, passed 1 test.
Prelude Test.QuickCheck> quickCheck (1 + 2 == 8)
*** Failed! Falsified (after 1 test):
```

Testable

Instanță interesantă

```
quickCheck :: Testable prop => prop -> IO ()

instance (Arbitrary a, Show a, Testable prop) =>
  Testable (a -> prop) where ..
```

- putem defini proprietăți care depind de parametri
- aproape toate tipurile standard de date sunt instanțe ale lui `Arbitrary`

```
Prelude> quickCheck (\x -> x + 0 == x)
+++ OK, passed 100 tests.
Prelude> quickCheck (\x y z -> (x + y) + z == x + (y + z))
+++ OK, passed 100 tests.
Prelude> quickCheck (\x y z -> (x - y) - z == x - (y - z))
*** Failed! Falsified (after 3 tests and 2 shrinks):
0
0
1
```

Testare QuickCheck

instance Testable Property where ...

Property ne poate ajuta să extindem limbajul logic cu combinatori precum:

- (==>) :: Testable prop => **Bool** –> prop –> Property

```
> quickCheck (\x y-> y /= 0 ==> x == y * (x `div` y) + x
      `mod` y)
+++ OK, passed 100 tests; 12 discarded.
```

Proprietatea definită de implicație respectă regulile implicației logice: testul reușește dacă premisa este falsă sau dacă rezultatul este adevărat. În cazul în care premisa este falsă testul este "aruncat" (discarded).

Testare QuickCheck

instance Testable Property where ...

Property ne poate ajuta să extindem limbajul logic cu combinatori precum:

- (==>) :: Testable prop => **Bool** -> prop -> Property
- (===) :: (**Eq** a, **Show** a) =>a -> a -> Property

```
> quickCheck (\x y-> x === y * (x `div` y) + x `mod` y)
*** Failed! Exception: 'divide by zero' (after 1 test):
0
0
Exception thrown while showing test case: 'divide by
    zero'
```

(===) se comportă ca (==) dar indică un contraexemplu

Testare QuickCheck

instance Testable Property where ...

Property ne poate ajuta să extindem limbajul logic cu combinatori precum:

- (==>) :: Testable prop => **Bool** -> prop -> Property
- (===) :: (**Eq** a, **Show** a) =>a -> a -> Property

```
> quickCheck (\x y-> x === y * (x `div` y) + x `mod` y)
*** Failed! Exception: 'divide by zero' (after 1 test):
0
0
Exception thrown while showing test case: 'divide by
    zero'
```

  (===) se comportă ca (==) dar indică un contraexemplu

- Mai sunt și alți combinatori: https://hackage.haskell.org/package/QuickCheck-2.14.2/docs/Test-QuickCheck.html#g:17

Testare QuickCheck

instance (Testable a) => Testable (Maybe a) where ...

- Ca și ==> de la Property ne poate ajuta să filtrăm cazurile nedefinite

```
testDivMod :: Integer -> Integer -> Maybe Bool
testDivMod _ 0 = Nothing
testDivMod x y = Just $ x == y * (x `div` y) + x `mod` y

Prelude Test.QuickCheck> quickCheck testDivMod
+++ OK, passed 100 tests; 11 discarded.
```

- Poate fi folosit cu orice Testable

```
testDivMod :: Integer -> Maybe (Integer -> Property)
testDivMod 0 = Nothing
testDivMod y =
  Just $ \x -> x === y * (x `div` y) + x `mod` y

Prelude Test.QuickCheck> quickCheck testDivMod
+++ OK, passed 100 tests; 15 discarded.
```

Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Tipuri de date algebrice

Tipurile de date algebrice se definesc folosind "operațiile" sumă și produs.

Forma generală

$$\begin{aligned} \mathit{data}\ \mathit{Typename} \quad = \quad & \mathit{Cons}_1\ \ t_{11}\dots t_{1k_1} \\ & \mid \mathit{Cons}_2\ \ t_{21}\dots t_{2k_2} \\ & \mid \dots \\ & \mid \mathit{Cons}_n\ \ t_{n1}\dots t_{nk_n} \end{aligned}$$

unde $k_1, \dots, k_n \geq 0$

- Se pot folosi tipuri sumă și tipuri produs.
- Se pot defini tipuri parametrizate.
- Se pot folosi definiții recursive.

Derivare automata vs Instanțiere explicită

- O clasă de tipuri este determinată de o mulțime de funcții.

```
class  Eq a  where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- minimum definition: (==)
  x /= y = not (x == y)
```

- Tipurile care aparțin clasei sunt instanțe ale clasei.
- Instanțierea prin derivare automată:

```
data Point a b = Pt a b
                 deriving Eq
```

- Instanțiere explicită:

```
instance Eq a => Eq (Point a b) where
      (==) (Pt x1 y1) (Pt x2 y2) = (x == x1)
```

Tipul Maybe (opțiune)

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

Argumente opționale

```haskell
power :: Maybe Int -> Int -> Int
power Nothing n    = 2 ^ n
power (Just m) n = m ^ n
```

Rezultate opționale

```haskell
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n `div` m)
```

Maybe - folosirea unui rezultat opțional

```haskell
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n `div` m)

-- utilizare gresita
wrong :: Int -> Int -> Int
wrong n m = divide n m + 3

-- utlizare corecta
right :: Int -> Int -> Int
right n m = case divide n m of
              Nothing -> 3
              Just r -> r + 3
```

A sau B

```haskell
data Either a b = Left a | Right b
```

```haskell
mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Definiți o funcție care întoarce concatenarea elementelor de tip **String**.

```haskell
addstrs    :: [Either Int String] -> String
addstrs    []                  = ""
addstrs    (Left n : xs)       = addstrs xs
addstrs    (Right s : xs) = s ++ addstrs xs
```

```haskell
addstrs' :: [Either Int String] -> String
addstrs' xs = concat [s | Right s <- xs]
```

Propoziții

Dorim să definim în Haskell calculul propozițional clasic.

```
type Name = String

data Prop = Var Name
          | F
          | T
          | Not Prop
          | Prop :|: Prop
          | Prop :&: Prop
          deriving (Eq, Ord)

type Names = [Name]
type Env = [(Name,Bool)] -- evaluarea variabilelor
```

Afișarea unei propoziții

```
showProp :: Prop -> String
showProp (Var x)    = x
showProp F          = "F"
showProp T          = "T"
showProp (Not p)    = par ("~" ++ showProp p)
showProp (p :|: q) = par (showProp p ++ "|" ++ showProp q)
showProp (p :&: q) = par (showProp p ++ "&" ++ showProp q)

par :: String -> String
par s = "(" ++ s ++ ")"

instance Show Prop where
   show = showProp
```

Mulțimea variabilelor unei propoziții

```
prop :: Prop
prop = (Var "a" :&: Not (Var "b"))

> names prop
["a","b"]

names    :: Prop ->    Names
names    (Var x)     = [x]
names    F           = []
names    T           = []
names    (Not p)     = names p
names    (p :|: q)   = nub (names p ++ names q)
names    (p :&: q)   = nub (names p ++ names q)
```

```
Prelude> :m + Data.List
Prelude Data.List> nub [1,2,2,3,1,4,2]
[1,2,3,4]
-- elimina duplicatele
```

Evaluarea unei propoziții

```
type Env = [(Name,Bool)]  -- evaluarea variabilelor
eval    :: Env -> Prop -> Bool
lookUp  :: Eq a => [(a,b)] -> a -> b

lookUp env x = head [ y | (x',y) <- , x == x' ]
-- nu tratam cazurile de eroare

eval  e (Var x)       = lookUp e x
eval  e F             = False
eval  e T             = True
eval  e (Not p)       = not (eval e p)
eval  e (p :|: q)     = eval e p || eval e q
eval  e (p :&: q)     = eval e p && eval e q
```

Propoziții

Exemple

```
p0 :: Prop
p0 = (Var "a" :&: Not (Var "a"))

e0 :: Env
e0 = [("a",True)]

*Main> showProp p0
"(a&(~a))"

*Main> names p0
["a"]

*Main> eval e0 p0
False

*Main> lookUp e0 "a"
True
```

Cum funcționează evaluarea?

```
    eval e0 (Var "a" :&: Not (Var "a"))
=
    (eval e0 (Var "a")) && (eval e0 (Not (Var "a")))
=
    (lookup e0 "a") && (eval e0 (Not (Var "a")))
=
    True && (eval e0 (Not (Var "a")))
=
  True && (not (eval e0 (Var "a")))
= ... =
  True && False
=
  False
```

Propoziții

Alte exemple

```
p1 :: Prop
p1 = (Var "a" :&: Var "b") :|:
       (Not (Var "a") :&: Not (Var "b"))

e1 :: Env
e1 = [("a",False), ("b",False)]

*Main> showProp p1
"((a&b)|((~a)&(~b)))"

*Main> names p1
["a","b"]

*Main> eval e1 p1
True

*Main> lookUp e1 "a"
False
```

Generarea tuturor evaluărilor

```
envs :: Names -> [Env]
envs []      = [[]]
envs (x:xs)  = [ (x,False):e | e <- envs xs ] ++
               [ (x,True ):e | e <- envs xs ]
```

Alternativă

```
envs :: Names -> [Env]
envs []      = [[]]
envs (x:xs) = [ (x,b):e | b <- bs, e <- envs xs ]
  where
  bs = [False,True]
```

Evaluări

```
  envs []  = [[]]

  envs ["b"]
= [("b",False):[]] ++ [("b",True ):[]]
= [[("b",False)], [("b",True )]]

  envs ["a","b"]
= [("a",False):e | e <- envs ["b"] ] ++
  [("a",True ):e | e <- envs ["b"] ]
= [("a",False):[("b",False)],("a",False):[("b",True )]] ++
  [("a",True ):[("b",False)],("a",True ):[("b",True )]]
= [[("a",False),("b",False)], [("a",False),("b",True )],
   [("a",True ),("b",False)],  [("a",True ),("b",True )]]
```

Exercițiu: Scrieți o funcție care verifică dacă o propoziție este satisfiabilă.

```
satisfiable :: Prop -> Bool
```

Expresii

```
data Exp      =   Lit Int
                | Add Exp Exp
                | Mul Exp Exp


showExp     :: Exp -> String
showExp  (Lit n)     = show n
showExp  (Add e1 e2) = par (showExp e1 ++ "+" ++ showExp
    e2)
showExp  (Mul e1 e2) = par (showExp e1 ++ "*" ++ showExp
    e2)

par :: String -> String
par s = "(" ++ s ++ ")"


instance Show Exp where
   show = showExp
```

Expresii

```haskell
data Exp = Lit Int
         | Add Exp Exp
         | Mul Exp Exp
```

Scrieți o funcție care evaluează expresiile:

```haskell
> evalExp $ Add (Lit 2) (Mul (Lit 3) (Lit 3))
11
```

```haskell
evalExp :: Exp -> Int
evalExp (Lit n)     = n
evalExp (Add e1 e2) = evalExp e1 + evalExp e2
evalExp (Mul e1 e2) = evalExp e1 * evalExp e2
```

Expresii

Exemple

```
ex0, ex1 :: Exp
ex0 = Add (Lit 2) (Mul (Lit 3) (Lit 3))
ex1 = Mul (Add (Lit 2) (Lit 3)) (Lit 3)

*Main> showExp ex0
"(2+(3*3))"

*Main> evalExp ex0
11

*Main> showExp ex1
"((2+3)*3)"

*Main> evalExp ex1
15
```

Expresii cu operatori

```
data    Exp    =    Lit Int
               |    Exp :+: Exp
               |    Exp :*: Exp

evalExp :: Exp -> Int
evalExp (Lit n)    = n
evalExp (e :+: f) = evalExp e + evalExp f
evalExp (e :*: f) = evalExp e * evalExp f

showExp :: Exp -> String
showExp (Lit n)    = show n
showExp (e :+: f) = par (showExp e ++ "+" ++ showExp f)
showExp (e :*: f) = par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s = "(" ++ s ++ ")"
```

Expresii cu operatori

Exemple

```
e0, e1 :: Exp
e0 = Lit 2 :+: (Lit 3 :*: Lit 3)
e1 = (Lit 2 :+: Lit 3) :*: Lit 3

*Main> showExp e0
"(2+(3*3))"

*Main> evalExp e0
11

*Main> showExp e1
"((2+3)*3)"

*Main> evalExp e1
15
```

Programare declarativă

Functori și categorii

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Tipuri parametrizate — „cutii”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „cutii”, recipiente care pot conține elemente de tipul dat ca argument.

Exemple

- Clasa de tipuri opțiune asociază unui tip a, tipul **Maybe** a
    - cutii goale: **Nothing**
    - cutii care țin un element x de tip a: **Just** x
- Clasa de tipuri listă asociază unui tip a, tipul [a]
    - cutii care țin 0, 1, sau mai multe elemente de tip a: [1, 2, 3], [], [5]

Tipuri parametrizate — „cutii”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „cutii”, recipiente care pot conține elemente de tipul dat ca argument.

Exemplu: tip de date pentru arbori binari

```
data Arbore a = Nil
              | Nod a Arbore Arbore
```

- Un arbore este o „cutie” care poate ține 0, 1, sau mai multe elemente de tip a:
  Nod 3 Nil (Nod 4 (Nod 2 Nil Nil) Nil), Nil, Nod 3 Nil Nil

Generalizare: Tipuri parametrizate — „computații”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „contexte computaționale”: computații care, atunci când se execută, pot produce rezultate de tipul dat ca argument.

Exemple

- **Maybe** a descrie rezultate de computații deterministe care pot eșua
  - computații care eșuează: **Nothing**
  - computații care produc un element de tipul dat: **Just** 4
- [**Int**] descrie liste de rezultate posibile ale unor computații nedeterministe
  - care pot produce oricare dintre rezultatele date: [1, 2, 3], [], [5]

Tipuri parametrizate — „computații”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „contexte computaționale”: computații care, atunci când se execută, pot produce rezultate de tipul dat ca argument.

Exemple

- **Either** e a descrie rezultate de tip a ale unor computații deterministe care pot eșua cu o eroare de tip *e*
  - **Right** 5 :: **Either** e **Int** reprezintă rezultatul unei computații reușite
  - **Left** "OOM":: **Either String** a reprezintă o excepție de tip **String**

Tipuri parametrizate — „computații”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „contexte computaționale”: computații care, atunci când se execută, pot produce rezultate de tipul dat ca argument.

Exemplu: tipul funcțiilor de sursă dată

- `t -> a` descrie computații care atunci când primesc o intrare de tip `t` produc un rezultat de tip `a`
  - `(++ "!")` :: **String** –> **String** este o computație care dat fiind un șir, îi adaugă un semn de exclamare
  - **length** :: **String** –> **Int** este o computație care dat fiind un șir, îi prduce lungimea acestuia
  - **id** :: **String** –> **String** este o computație care produce șirul dat ca argument

Clase de tipuri pentru cutii și computații?

Întrebare

Care sunt trăsăturile comune ale acestor tipuri parametrizate care pot fi gândite intuitiv ca cutii care conțin elemente / computații care produc rezultate?

Problemă

Putem proiecta clase de tipuri care descriu funcționalități comune tuturor acestor tipuri?

Problemă

Formulare cu cutii

Dată fiind o funcție f :: a −> b și o cutie ca care conține elemente de tip a, vreau să să obțin o cutie cb care conține elemente de tip b obținute prin transformarea elementele din cutia ca folosind funcția f (și doar atât!)

Formulare cu computații

Dată fiind o funcție f :: a −> b și o computație ca care produce rezultate de tip a, vreau să să obțin o computație cb care produce rezultate de tip b obținute prin transformarea rezultatelor produse de computația ca folosind funcția f (și doar atât!)

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 (și doar atât!)

Clasa de tipuri Functor

Definiție

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

Dată fiind o funcție f :: a -> b și ca :: m a, fmap produce cb :: m b obținută prin transformarea rezultatelor produse de computația ca folosind funcția f (și doar atât!)

Instanță pentru liste

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

Clasa de tipuri Functor

Instanțe

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

Instanță pentru tipul arbore fmap :: (a -> b) -> Arbore a -> Arbore b

```
instance Functor Arbore where
  fmap f Nil = Nil
  fmap f (Nod x l r) = Nod (f x) (fmap f l) (fmap f r)
```

Clasa de tipuri Functor

Instanțe

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

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

```haskell
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)

```haskell
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
```

Proprietăți ale functorilor

- Argumentul m al lui **Functor** m definește o transformare de tipuri
  - m a este tipul a transformat prin functorul m
- fmap definește transformarea corespunzătoare a funcțiilor
  - fmap :: (a -> b) ->(m a -> m b)

Contractul lui fmap

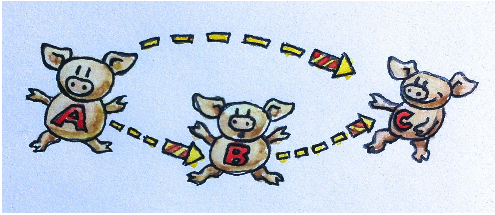
- fmap f ca e obținută prin transformarea rezultatelor produse de computația ca folosind funcția f (și doar atât!)
- Abstractizat prin două legi:

identitate	fmap **id** == **id**
compunere	fmap (g . f) == fmap g . fmap f

Categorii

O categorie $\mathbb{C}$ este dată de:

- O clasă $|\mathbb{C}|$ a obiectelor
- Pentru oricare două obiecte $A, B \in |\mathbb{C}|$,
  o mulțime $\mathbb{C}(A, B)$ a săgeților „de la A la B”
  $f \in \mathbb{C}(A, B)$ poate fi scris ca $f : A \to B$
- Pentru orice obiect A o săgeată $id_A : A \to A$ numită identitatea lui A
- Pentru orice obiecte A, B, C, o operație de compunere a săgeților
  $\circ : \mathbb{C}(B, C) \times \mathbb{C}(A, B) \to \mathbb{C}(A, C)$



Bartosz Milewski — Category: The Essence of Composition

- Compunerea este asociativă și are element neutru id

Exemplu: Categoria $\mathbb{S}\mathrm{et}$

- Obiecte: mulțimi
- Săgeți: funcții
- Identități: Funcțiile identitate
- Compunere: Compunerea funcțiilor

Exemplu: Categoria $\mathbb{H}$ask

- Obiectele: tipuri
- Săgețiile: funcții între tipuri

```
f :: A -> B
```

- Identități: funcția polimorfică **id**

```
Prelude> :t id
id :: a -> a
```

- Compunere: funcția polimorfică (.)

```
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Subcategorii ale lui Hask date de tipuri parametrizate

- Obiecte: o clasă restânsă de tipuri din $|\mathbb{H}\mathrm{ask}|$
  - Exemplu: tipuri de forma [a]
- Săgeți: toate funcțiile din $\mathbb{H}$ask între tipurile obiecte
  - Exemple: **concat** :: [[a]] −> [a], **words** :: [**Char**] −> [**String**], **reverse** :: [a] −> [a]

Exemple

Liste	obiecte: tipuri de forma [a]
Optiuni	obiecte: tipuri de forma Maybe a
Arbori	obiecte: tipuri de forma Arbore a
Funcții de sursă t	obiecte: tipuri de forma t -> a

De ce categorii?

(Des)compunerea este esența programării

- Am de rezolvat problema P
- O descompun în subproblemele $P_1, \dots P_n$
- Rezolv problemele $P_1, \dots P_n$ cu programele $p_1, \dots p_n$
  - Eventual aplicând recursiv procedura de față
- Compun rezolvările $p_1, \dots p_n$ într-o rezolvare p pentru problema inițială

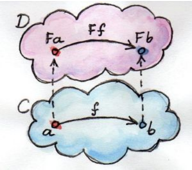
Categoriile rezolvă problema compunerii

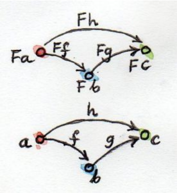
- Ne forțează să abstractizăm datele
- Se poate acționa asupra datelor doar prin săgeți (metode?)
- Forțează un stil de compunere independent de structura obiectelor

Functori

Date fiind două categorii $\mathbb{C}$ și $\mathbb{D}$, un functor $F : \mathbb{C} \to \mathbb{D}$ este dat de

- O funcție $F : |\mathbb{C}| \to |\mathbb{D}|$ de la obiectele lui $\mathbb{C}$ la cele ale lui $\mathbb{D}$
- Pentru orice $A, B \in |\mathbb{C}|$, o funcție $F : \mathbb{C}(A, B) \to \mathbb{D}(F(A), F(B))$
- Compatibilă cu identitățile și cu compunerea
  - $F(id_A) = id_{F(A)}$ pentru orice A
  - $F(g \circ f) = F(g) \circ F(f)$ pentru orice $f : A \to B, g : B \to C$, $h = g \circ f$





Bartosz Milewski — Functors

Functori în Haskell

În general un functor $F : \mathbb{C} \to \mathbb{D}$ este dat de

- O funcție $F : |\mathbb{C}| \to |\mathbb{D}|$ de la obiectele lui $\mathbb{C}$ la cele ale lui $\mathbb{D}$
- Pentru orice $A, B \in |\mathbb{C}|$, o funcție $F : \mathbb{C}(A, B) \to \mathbb{D}(F(A), F(B))$
- Compatibilă cu identitățile și cu compunerea
  - $F(id_A) = id_{F(A)}$ pentru orice A
  - $F(g \circ f) = F(g) \circ F(f)$ pentru orice $f : A \to B, g : B \to C, h = g \circ f$

În Haskell o instanță **Functor** m este dată de

- Un tip m a pentru orice tip a (deci m trebuie sa fie tip parametrizat)
- Pentru orice două tipuri a și b, o funcție

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

- Compatibilă cu identitățile și cu compunerea

```
fmap id == id
fmap (g . f) == fmap g . fmap f
```

pentru orice f :: a -> b și g :: b -> c

Programare declarativă

Monoid, Foldable

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

din nou **foldr**

```
foldr :: (a -> b -> b) -> b -> t a -> b
```

```
Prelude> foldr (+) 0 [1,2,3]
6
Prelude> foldr (*) 1 [1,2,3]
6
Prelude> foldr (++) [] ["1","2","3"]
"123"
Prelude> foldr (||) False [True, False, True]
True
Prelude> foldr (&&) True [True, False, True]
False
```

Ce au in comun aceste operații?

Monoizi

$(M, \circ, e)$ este monoid dacă
$\circ : M \times M \to M$ este asociativă
$m \circ e = e \circ m = m$ oricare $m \in M$

Exemple de monoizi

(Int, +,0), (Int, $*$, 1), (String, ++, []), ({True,False}, &&, True), ({True,False}, ||, False)

Operația de monoid poate fi generalizată pe liste:

```
sum = foldr (+) 0
product = foldr (*) 1
concat = foldr (++) []
and = foldr (&&) True
or = foldr (||) False
```

Monoizi și semigrupuri

Monoid

(M,$\circ$, e) este monoid dacă
$\circ : M \times M \rightarrow M$ este asociativă
$m \circ e = e \circ m = m$ oricare $m \in M$

Un semigrup este un monoid fără element neutru

(M,$\circ$) este monoid dacă
$\circ : M \times M \rightarrow M$ este asociativă

Exemple

- Orice monoid este și semigrup
- Semigrupul numerelor naturale pozitive, cu adunarea $(\mathbb{N}^*, +)$
- Semigrupul numerelor intregi nenule, cu înmulțirea $(\mathbb{Z}^*, *)$
- Semigrupul listelor nevide, cu concatenarea

clasele **Semigroup** și **Monoid**

https://hackage.haskell.org/package/base/docs/Prelude.html#t:Semigroup

```
class Semigroup a where
  (<>) :: a -> a -> a       -- operatia asociativa
infixr 6 <>

class Semigroup a => Monoid a where
    mempty  :: a                 -- elementul neutru

    mconcat :: [a] -> a          -- generalizarea la liste
    mconcat = foldr (<>) mempty
```

Legi

- Asociativitate: x <> (y <> z) = (x <> y) <> z
- Identitate la dreapta: x <> mempty = x
- Identitate la stânga: mempty <> x = x
- Atenție! Acest lucru este responsabilitatea programatorului!

clasa **Monoid**

Exemple

Listele ca instanța

```
instance Semigroup [a] where
    (<>) = (++)
instance Monoid [a] where
    mempty = []

Prelude> mempty :: [a]
[]
Prelude> mconcat [[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

Mai multe instanțe pentru același tip?

(Int, $+$,0), (Int, $*$, 1) sunt monoizi

({True,False}, &&, True), ({True,False}, ||, False) sunt monoizi

Problemă: Cum definim instante diferite pentru acelasi tip?

clasa **Monoid**

(Int, $+$,0), (Int, $*$, 1) sunt monoizi
({True,False}, &&, True), ({True,False}, ||, False) sunt monoizi

Cum definim instante diferite pentru acelasi tip?

- se crează o copie a tipului folosind **newtype**
- copia este definită ca instanță a tipului

newtype

```
newtype Nat = MkNat Integer
```

- **newtype** se folosește cînd un singur constructor este aplicat unui singur tip de date
- declarația cu **newtype** este mai eficientă decât cea cu **data**
- **type** redenumește tipul; **newtype** face o copie și permite redefinirea operațiilor

clasa **Monoid**

All și Any

- **Bool** ca monoid față de conjuncție

```haskell
newtype All = All { getAll :: Bool }
    deriving (Eq, Read, Show)

instance Semigroup All where
    All x <> All y = All (x && y)
instance Monoid All where
    mempty = All True
```

- **Bool** ca monoid față de disjuncție

```haskell
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Read, Show)

instance Semigroup Any where
    Any x <> Any y = Any (x || y)
instance Monoid Any where
    mempty = Any False
```

clasa **Monoid**

Sum și Product

- **Num a** ca monoid față de adunare

```
newtype Sum a = Sum { getSum :: a }
    deriving (Eq, Read, Show)

instance Num a => Semigroup (Sum a) where
    Sum x <> Sum y = Sum (x + y)
instance Num a => Monoid (Sum a) where
    mempty = Sum 0
```

- **Num a** ca monoid față de înmulțire

```
newtype Product a = Product { getProduct :: a }
    deriving (Eq, Read, Show)

instance Num a => Semigroup (Product a) where
    Product x <> Product y = Product (x * y)
instance Num a => Monoid (Product a) where
    mempty = Product 1
```

clasa **Monoid**

Min și Max

- **Ord a** ca semigrup față de operația de minim

```haskell
newtype Min a = Min { getMin :: a }
    deriving (Eq, Read, Show)

instance Ord a => Semigroup (Min a) where
    Min x <> Min y = Min (min x y)
instance (Ord a, Bounded a) => Monoid (Min a) where
    mempty = Min maxBound
```

- **Ord a** ca semigrup față de operația de maxim

```haskell
newtype Max a = Max { getMax :: a }
    deriving (Eq, Read, Show)

instance Ord a => Semigroup (Max a) where
    Max x <> Max y = Max (max x y)
instance (Ord a, Bounded a) => Monoid (Max a) where
    mempty = Max minBound
```

clasa **Monoid**

Exemple

```
Prelude> Sum 3
<interactive>:15:1: error:

Prelude> :m + Data.Monoid
Prelude Data.Monoid> Sum 3
Sum {getSum = 3}
Prelude Data.Monoid> Sum 3 <> Sum 4
Sum {getSum = 7}
Prelude Data.Monoid> Product 3 <> Product 4
Product {getProduct = 12}
Prelude Data.Monoid> mconcat [Any False,Any True,Any False]
Any {getAny = True}
Prelude Data.Monoid> (getSum . mconcat) [Sum 3,Sum 4,Sum 5]
12
Prelude Data.Monoid> getMax . mconcat . map Product $
    [3,5,4]
5
```

Monoid Maybe

```
instance Semigroup a => Semigroup (Maybe a) where
    Nothing <> m        =  m
    m       <> Nothing  =  m
    Just m1 <> Just m2  =  Just (m1 <> m2)

instance Semigroup a => Monoid (Maybe a) where
    mempty =  Nothing
```

```
Prelude Data.Monoid> Nothing <> (Just 3) :: Maybe Integer
<interactive>:35:1: error:

Prelude Data.Monoid> Nothing <> (Just (Sum 3))
Just (Sum {getSum = 3})
```

Funcții ca instanțe

(a -> a) ca instanța a clasei **Monoid**

```
newtype Endo a = Endo { appEndo :: a -> a }

instance Monoid Endo where
     mempty            = Endo id
     Endo g <> Endo f = Endo (g . f)

Prelude> :m + Data.Monoid
>let f = mconcat [Endo (+1), Endo (+2), Endo (+3)]
>:t f
f :: Num a => Endo a

> (appEndo f) 0
6
> (appEndo . mconcat) [Endo (+1), Endo (+2), Endo (+3)] $ 0
6
```

Semigroup

NonEmpty

Tipul listelor nevide

```
data NonEmpty a = a :| [a]      deriving (Eq, Ord)

instance Semigroup (NonEmpty a) where
    (a :| as) <> (b :| bs) = a :| (as ++ b : bs)
```

Concatenare pentru semigrupuri

```
sconcat :: Semigroup a => NonEmpty a -> a
sconcat (a :| as) = go a as
  where
    go a [] = a
    go a (b : bs) = a <> go b bs
```

din nou **foldr**

foldr pe liste

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f i [] = i
foldr f i (x:xs) = f x (foldr f i xs)
```

Problema: să generalizăm **foldr** la alte structuri recursive.

Exemplu: arbori binari

```
data BinaryTree a = Leaf a
                  | Node (BinaryTree a) (BinaryTree a)
                  deriving Show
```

Cum definim **"foldr"** înlocuind listele cu date de tip **BinaryTree** ?

"foldr" folosind BinaryTree

```
data BinaryTree a = Leaf a
                  | Node (BinaryTree a) (BinaryTree a)
                  deriving Show
```

foldTree

```
foldTree :: (a -> b -> b) -> b -> BinaryTree a -> b
foldTree f i (Leaf x) = f x i
foldTree f i (Node l r) = foldTree f (foldTree f i r) l
```

foldTree

```
data BinaryTree a = Leaf a
                  | Node (BinaryTree a) (BinaryTree a)
                  deriving Show

foldTree :: (a -> b -> b) -> b -> BinaryTree a -> b
foldTree f i (Leaf x) = f x i
foldTree f i (Node l r) = foldTree f (foldTree f i r) l

myTree = Node (Node (Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))

*Main> foldTree (+) 0 myTree
10
```

clasa **Foldable**

https://en.wikibooks.org/wiki/Haskell/Foldable
https://hackage.haskell.org/package/base/docs/Data-Foldable.html

Data.Foldable

```
class Foldable t where
        fold    :: Monoid m => t m -> m
        foldMap :: Monoid m => (a -> m) -> t a -> m
        foldr   :: (a -> b -> b) -> b -> t a -> b

        fold = foldMap id
        ...
```

Observații:

- definiția minimală completă conține fie **foldMap**, fie **foldr**
- **foldMap** și **foldr** pot fi definite una prin cealaltă
- pentru a crea o instanță este suficient să definim una dintre **foldMap** și **foldr**, cealaltă va fi automat accesibilă

Foldable cu foldr

```
instance Foldable BinaryTree where
   foldr = foldTree

treeI = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
treeS = Node (Node(Leaf "1")(Leaf "2"))
             (Node (Leaf "3")(Leaf "4"))

*Main> foldr (+) 0 treeI
10
*Main> foldr (++) [] treeS
"1234"
```

clasa **Foldable**

Data.Foldable

```
class Foldable t where
        fold    :: Monoid m => t m -> m
        foldMap :: Monoid m => (a -> m) -> t a -> m
        foldr   :: (a -> b -> b) -> b -> t a -> b

        fold = foldMap id
        ...
```

```
instance Foldable BinaryTree where
    foldr = foldTree
```

Observație: în definiția clasei **Foldable**, variabila de tip t nu reprezintă un tip concret ([a], Sum a) ci un constructor de tip (BinaryTree)

Foldable cu foldr

```
instance Foldable BinaryTree where
   foldr = foldTree

treeI = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
treeS = Node (Node(Leaf "1")(Leaf "2"))
             (Node (Leaf "3")(Leaf "4"))
```

Avem definite automat **foldMap** și alte funcții precum: **foldl, foldr',foldr1,...**

```
*Main> foldl (++) [] treeS
"1234"
*Main> foldl (+) 0 treeI
10
*Main> maximum treeI
4

*Main Data.Monoid> foldMap Sum treeI
Sum {getSum = 10}
*Main Data.Monoid> foldMap id treeS
"1234"
```

foldMap

```haskell
foldMap :: Monoid m => (a -> m) -> t a -> m
```

```haskell
newtype Sum a = Sum { getSum :: a }
                deriving (Eq, Read, Show)

instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    Sum x <> Sum y = Sum (x + y)

treeI = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))

*Main> foldMap Sum treeI    -- Sum :: a -> Sum a
Sum {getSum = 10}
```

sum cu foldMap

```
foldMap :: Monoid m => (a -> m) -> t a -> m

newtype Sum a = Sum { getSum :: a }
                deriving (Eq, Read, Show)

instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    Sum x <> Sum y = Sum (x + y)

sum as = getSum $ foldMap Sum as
sum = getSum . (foldMap Sum)

treeI = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))

*Main> foldMap Sum treeI    -- Sum :: a -> Sum a
Sum {getSum = 10}
*Main> sum treeI
 10
```

product cu foldMap

```
foldMap :: Monoid m => (a -> m) -> t a -> m

newtype Product a = Product { getProduct :: a }
    deriving (Eq, Read, Show)

instance Num a => Semigroup (Product a) where
    Product x <> Product y = Product (x * y)
instance Num a => Monoid (Product a) where
    mempty = Product 1

product as = getProduct$ foldMap Product as
product = getProduct . (foldMap Product)

treeI = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))

*Main> foldMap Product treeI
Product {getProduct = 24}
*Main> product treeI
 24
```

elem cu foldMap

```
foldMap :: Monoid m => (a -> m) -> t a -> m
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Read, Show)
instance Semigroup Any where
    Any x <> Any y = Any (x || y)
instance Monoid Any where
    mempty = Any False

any as = getAny $ foldMap Any as
any = getAny . (foldMap Any)

elem e = getAny . (foldMap (Any . (== e)))

treeI = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))

*Main> foldMap (Any . (== 1)) treeI
Any {getAny = True}
*Main> elem 1  treeI
 True
```

foldMap folosind **foldr**

http://cmsc-16100.cs.uchicago.edu/2016/Lectures/13-monoid-foldable.php

Cum definim **foldMap** folosind **foldr**?

```
foldr    :: (a -> b -> b) -> b -> t a -> b
foldMap  :: Monoid m => (a -> m) -> t a -> m
```

```
foldMap  f tr = foldr foo i tr      -- f   :: a -> m
                where foo  = ???    -- foo  :: (a -> m -> m)
                      i  = mempty
```

```
foo = \x acc -> f x <> acc
    = \x acc -> (<>) (f x) acc
    = \x -> (<>) $ f x
    = \x -> ((<>) . f) x
    = (<>) . f
```

foldMap f = foldr ((<>) . f) mempty

Foldable cu foldMap

```
instance Foldable BinaryTree where
    foldMap f (Leaf x)    = f x
    foldMap f (Node l r) = foldMap f l <> foldMap f r

treeI = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
treeS = Node (Node(Leaf "1")(Leaf "2"))
             (Node (Leaf "3")(Leaf "4"))
```

Avem definite automat **foldr** și alte funcții precum: **foldl, foldr',foldr1,...**

```
*Main> foldr (++) [] treeS
"1234"
*Main> foldl (+) 0 treeI
10
```

foldr folosind **foldMap**

https://en.wikibooks.org/wiki/Haskell/Foldable

Cum definim **foldr** folosind **foldMap**?

```
foldr   :: (a -> b -> b) -> b -> t a -> b
foldMap :: Monoid m => (a -> m) -> t a -> m
```

Idee

```
foldr   :: (a -> (b -> b)) -> b -> t a -> b
```

- pentru fiecare element de tip **a** din **t a** se crează o funcție de tip **(b->b)**

  obținem, de exemplu, o lista de funcții sau
  un arbore care are ca frunze funcții

- folosim faptul ca **(b->b)** este instanță a lui **Monoid** și aplicăm **foldMap**

foldr folosind **foldMap**

https://en.wikibooks.org/wiki/Haskell/Foldable

```haskell
foldr    :: (a -> (b -> b)) -> b -> t a -> b
```

(b->b) instanță a lui **Monoid**

```haskell
newtype Endo b = Endo { appEndo :: b -> b }
instance Monoid Endo where
    mempty            = Endo id
    Endo g <> Endo f  = Endo (g . f)
```

Definim funcția ajutătoare

```haskell
foldComposing :: (a -> (b -> b)) -> t a -> Endo b
```

astfel încât

```haskell
foldr f i tr = appEndo (foldComposing f tr) $ i
```

foldr folosind **foldMap**

https://en.wikibooks.org/wiki/Haskell/Foldable

```
foldr          :: (a -> (b -> b)) -> b -> t a -> b
foldComposing  :: (a -> (b -> b)) -> t a -> Endo b

foldComposing f = foldMap (Endo . f)
```

Exemplu:

```
foldComposing (+) [1, 2, 3]
foldMap (Endo . (+)) [1, 2, 3]
(Endo . (+)) 1 <> (Endo . (+)) 2 <> (Endo . (+)) 3
Endo (+1) <> Endo (+2) <> Endo (+3)
Endo ((+1) . (+2) . (+3))
Endo (+6)
```

```
foldr f i tr = appEndo (foldComposing f tr) $ i
```

Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Finger trees

https://apfelmus.nfshost.com/articles/monoid-fingertree.html

- implementarea eficientă și unitară a structurilor de date funcționale
- informația se află în frunzelor
- nodurilor interne conțin valori a căror structură determină funcționalitatea arborelui

```
       v
      / \
    v     v
   /\     /\
  v   v v   v
  a   a a  /\
          v   v
          a   a
```

Finger Tree

```
data  FTree v a =   Leaf v a
                  | Node v  (FTree v a) (FTree v a)
                  deriving Show
```

```
        v
       / \
      v   v
     /\   /\
    v  v v  v
    a  a a  /\
           v  v
           a  a
```

Finger Tree

```haskell
data  FTree v a =   Leaf v a
                  | Node v  (FTree v a) (FTree v a)
                  deriving Show

exFT = Node  5
           (Node 2
                (Leaf 1 'a')
                (Leaf 1 'b'))
           (Node 3
                (Leaf 1 'c')
                (Node 2
                     (Leaf 1 'd')
                     (Leaf 1 'e')))


*Main> :t exFT
exFT :: FTree Int String
```

Finger Tree

Ce reprezinta informatia din arbore?

```
        5
       / \
     2     3
    /\     /\
   1   1 1   2
   a   b c   /\
            1   1
            d   e
```

(P1) Valoarea fiecărui nod intern reprezintă numărul de elemente din arborele respectiv.

```haskell
type Size = Int
exFT :: FTree Size Char
```

Finger Tree

- (P1) Valoarea fiecărui nod intern reprezintă numărul de elemente din arborele respectiv.

```
tag :: FTree v a -> v
tag (Leaf n _) = n
tag (Node n _ _) = n
```

- Lista datelor este alcătuită din frunze.

```
toList :: FTree v a -> [a]
toList (Leaf _ a)     = [a]
toList (Node _ x y) = toList x ++ toList y
```

Finger Tree

```
data  FTree v a =   Leaf v a
                  | Node v  (FTree v a) (FTree v a)
                  deriving Show
```

Construcția unui arbore cu (P1) se poate face cu funcții constructor specifice:

```
type Size = Integer

leaf :: a -> FTree Size a
leaf x = Leaf 1 x

node :: FTree Size a -> FTree Size a -> FTree Size a
node  t1 t2 = Node ((tag t1) + (tag t2)) t1 t2
```

```
exFT == node (node (leaf 'a')(leaf 'b'))
             (node (leaf 'c') (node (leaf 'd')(leaf 'e')))
```

Finger Tree

Accesarea elementului din poziția **n**

```
(!!!) :: FTree Size a -> Int -> a
(Leaf _ a)      !!! 0 = a
(Node _ x y)  !!! n
     | n < tag x       = x !!! n
     | otherwise       = y !!! (n - tag x)
```

```
*Main> exFT !!! 3
'd'
```

Dacă arborele se menține echilibrat, timpul de acces poate fi îmbunătățit.

Finger Tree

Priority Queue

(P2) Valoarea fiecărui nod intern reprezintă cea mai mica prioritate din arborele respectiv.

```
pqFT = Node  2
         (Node 4
              (Leaf 16 'a')
              (Leaf 4 'b'))
         (Node 2
              (Leaf 2 'c')
              (Node 8
                   (Leaf 32 'd')
                   (Leaf 8 'e')))
```

```
type Priority = Int
pqFT :: FTree Priority Char
```

Finger Tree

Construcția unui arbore cu (P2) se poate face cu funcții constructor specifice:

```
type Priority = Int

pleaf :: Priority -> a -> FTree Priority a
pleaf n x = Leaf n x

pnode :: FTree Priority a -> FTree Priority a -> FTree
    Priority a
pnode  t1 t2 = Node ((tag t1) `min` (tag t2)) t1 t2
```

```
exFT == pnode (pnode (pleaf 16 'a')(pleaf 4 'b'))
                (pnode (pleaf 2 'c') (pnode (pleaf 32 'd')(
                    pleaf 8 'e')))
```

Finger Tree

Priority Queue - determinarea elementului cîțtigător

```
winner :: FTree Priority a -> a
winner t = go t
    where
    go (Leaf _ a)         = a
    go (Node _ x y)
        | tag x == tag t = go x
        | tag y == tag t = go y
```

```
*Main> winner pqFT
'c'
```

Dacă arborele se menține echilibrat, timpul de acces poate fi îmbunătățit.

Finger Trees

Unificarea celor două exemple

Observăm că

- Pentru arbori cu (P1) funcția **tag** verifică:
  $tag :: FTree\ \ Size\, a \to Size$
  $tag(Leaf\ \_) = 1$
  $tag(Node\ \_\ x\, y) = \ tag\, x\ +\ tag\, y$

  $(Size, +, 0)$ monoid

- Pentru arbori cu (P2) funcția **tag** verifică:
  $tag :: FTree\ \ Priority\, a \to Priority$
  $tag(Leaf\ \_\ a) = priority\ \ a$
  $tag(Node\ \_\ x\, y) = tag\, x\, `min`\, tag\, y$

  $(Priority, min, maxBound)$ monoid

Finger Trees

Unificarea celor două exemple folosind monoizi

- Pentru arbori cu (P1) definim o instanța **Monoid** a lui **Size**:

```
instance Monoid Size where
    mempty  = 0
    mappend = (+)
```

- Pentru arbori cu (P2) definim o instanța **Monoid** a lui **Priority**:

```
instance Monoid Priority where
    mempty  = maxBound
    mappend = min
```

Atenție!

În acest exemplu **Size** și **Priority** sunt redenumiri ale lui **Int**. Pentru a putea fi făcute instanțe ale clasei **Monoid** simultan trebuie folosit **newtype**.

Finger Trees

Unificarea celor două exemple folosind monoizi

Constructorul pentru **Node**

```
node :: Monoid v => FTree v a -> FTree v a -> FTree v a
node x y = Node (tag x <> tag y) x y
```

Constructorul pentru **Leaf**

Cum transmitem tag-urile asociate frunzelor?

```
leaf :: Monoid v => (a->v) -> a -> FTree v a
leaf measure x = Leaf (measure x) x
```

Transmitem ca parametru o funcție care asociază fiecărei date tag-ul corespunzător.

Finger Trees

Unificarea celor două exemple folosind monoizi

Constructorii pentru **Node** și **Leaf**

```
node :: Monoid v => FTree v a -> FTree v a -> FTree v a
node x y = Node (tag x <> tag y) x y

leaf :: Monoid v => (a->v) -> a -> FTree v a
leaf measure x = Leaf (measure x) x
```

```
priority :: Char -> Int
*Main> leaf priority 'a'
Leaf 16 'a'
*Main> node (leaf priority 'a') (leaf priority 'b')
Node 3 (Leaf 16 'a') (Leaf 4 'b')
*Main> node (Leaf 16 'a') (Leaf 4 'b') :: FTree Int Char
Node 3 (Leaf 16 'a') (Leaf 4 'b')
```

Finger Trees

Unificarea căutării

```haskell
search :: Monoid v => (v -> Bool) -> FTree v a -> Maybe a
search  p t
    | p (tag t) = Just (go mempty p t)
    | otherwise       = Nothing
    where
    go i p (Leaf _ a) = a
    go i p (Node _ l r)
        | p (i <> tag l) = go i p l
        | otherwise               = go (i <> tag l) p r
```

Finger Trees

Unificarea căutării

Int ca Size

```
instance Monoid Int where
    mempty  = 0
    mappend = (+)

win k t = search (>= k) t
```

Int ca Priority

```
instance Monoid Int where
    mempty  = maxBound
    mappend = min       -- Int ca Priority

win t = search (== tag t) t
```

Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Tipuri de date algebrice

Tipurile de date algebrice se definesc folosind "operațiile" sumă și produs.

Forma generală

$$
\begin{aligned}
\mathit{data}\ \mathit{Typename} \quad = \quad & \mathit{Cons}_1\ \ t_{11} \ldots t_{1k_1} \\
& \mid \mathit{Cons}_2\ \ t_{21} \ldots t_{2k_2} \\
& \mid \ldots \\
& \mid \mathit{Cons}_n\ \ t_{n1} \ldots t_{nk_n}
\end{aligned}
$$

unde $k_1, \ldots, k_n \geq 0$

- Se pot folosi tipuri sumă și tipuri produs.
- Se pot defini tipuri parametrizate.
- Se pot folosi definiții recursive.

din nou **foldr**

Problema: să generalizăm **foldr** la alte structuri recursive.

```
data Exp   =   Lit Int
           |    Add Exp Exp
           |    Mul Exp Exp
```

Cum definim **"foldr"** înlocuind listele cu date de tip **Exp** ?

```
evalExp :: Exp -> Int
evalExp (Lit n)     = n
evalExp (Add e1 e2) = evalExp e1 + evalExp e2
evalExp (Mul e1 e2) = evalExp e1 * evalExp e2
```

Vrem să definim **"foldExp"** astfel încât

```
evalExp = foldExp fLit (+) (*)
```

din nou **foldr**

```
data Exp    =   Lit Int
              |   Add Exp Exp
              |   Mul Exp Exp



foldExp fLit fAdd fMul (Lit n)    = fLit  n
foldExp fLit fAdd fMul (Add e1 e2) = fAdd v1 v2
                    where
                          v1 = foldExp fLit fAdd fMul e1
                          v2 = foldExp fLit fAdd fMul e2

foldExp fLit fAdd fMul (Mul e1 e2) = fMul v1 v2
                     where
                          v1 = foldExp fLit fAdd fMul e1
                          v2 = foldExp fLit fAdd fMul e2

evalExp = foldExp fLit (+) (*)
          where  fLit (Lit x) = x
```

din nou **foldr**

```
data Exp   =   Lit Int
           |   Add Exp Exp
           |   Mul Exp Exp

foldExp fLit fAdd fMul (Lit n)    = fLit  n
foldExp fLit fAdd fMul (Add e1 e2) = fAdd v1 v2
                          where ...

foldExp fLit fAdd fMul (Mul e1 e2) = fMul v1 v2
                          where ...
```

Ce tip are **foldExp**?

```
foldExp :: (Int->b)->(b->b->b)->(b->b-> b)-> Exp Int->b
```

Programare declarativă[1]

Intrare/Ieșire

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

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

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

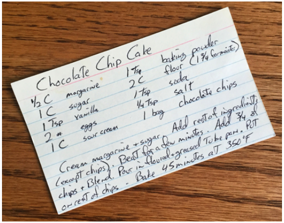
```
putChar :: Char -> IO ()
>    putChar '!'
```

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

Mind-Body Problem - Rețetă vs Prăjitură

http://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html

c :: Cake



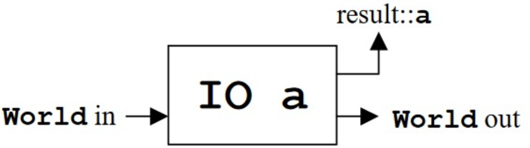
r :: Recipe Cake

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: ...

Combină două comenzi!

```haskell
(>>) :: IO () -> IO () -> IO ()
putChar :: Char -> IO ()
```

Exemplu

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

rerpezintă o comandă care, dacă va fi executată, va afișa un semn de întrebare urmat de un semn de exclamare .

Afișează un șir de caractere

```
putStr :: String -> IO ()
putStr []     = done
putStr (x:xs) = putChar x >> putStr xs
```

Observație:

```
done :: IO ()
```

reprezintă o comandă care, dacă va fi executată, nu va face nimic.

Exemplu

```
putStr "?!" == putChar '?' >> (putChar '!' >> done)
```

rerpezintă o comandă care, dacă va fi executată, va afișa un semn de întrebare urmat de un semn de exclamare.

putStr folosind funcționale

```
putStr      :: String -> IO ()
putStr      = foldr (>>) done . map putChar
```

Afișează și treci pe rândul următor

```
putStrLn :: String -> IO ()
putStrLn xs = putStr xs >> putChar '\n'
```

(IO (), (>>), done) e monoid

```
m >> done          =   m
done >> m          =   m
(m >> n) >> o      =   m >> (n >> o)
```

Când sunt executate comenzile?

main

Orice comandă **IO a** poate fi executată în interpretor, dar

Programele Haskell pot fi compilate

Fișierul scrie.hs:

```haskell
main :: IO ()
main = putStrLn "?!"
```

```
08-io$ ghc scrie.hs
[1 of 1] Compiling Main  ( scrie.hs, scrie.o)
Linking scrie.exe ...
08-io$ ./scrie
?!
```

Funcția executată este **main**

Raționamentele substitutive sunt valabile

În Haskell

Expresii

```
(1+2) * (1+2)
```

este echivalentă cu expresia

```
let x = 1+2 in x * x
```

și se evaluează amândouă la 9

Comenzi

```
putStr "HA!" >> putStr "HA!"
```

este echivalentă cu

```
let m = putStr "HA!" in m >> m
```

si amândouă afișează "HA!HA!".

Raționamentele substitutive sunt valabile

https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor

Referential transparency

orice expresie poate fi înlocuită cu valoare ei

```
addExclamation :: String -> String
addExclamation s = s ++ "!"

main = putStrLn (addExclamation "Hello")
Prelude> main
Hello!

main = putStrLn ("Hello" ++ "!")
Prelude> main
Hello!
```

Raționamentele substitutive sunt valabile

https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor

```haskell
addExclamation :: String -> String
addExclamation s = s ++ "!"
```

Observație

Dacă **getLine** ar avea tipul **String** atunci am putea scrie

```haskell
main = putStrLn (addExclamation getLine)  -- cod eronat !!!
```

Raționamentele substitutive sunt valabile

https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor

```haskell
addExclamation :: String -> String
addExclamation s = s ++ "!"
```

Observație

Dacă **getLine** ar avea tipul **String** atunci am putea scrie

```haskell
main = putStrLn (addExclamation getLine)  -- cod eronat !!!
```

Nu putem înlocui **getLine** cu valoarea ei!

Soluția: **getLine** are tipul **IO String**

Comenzi cu valori

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

- În general, **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"

Produ o valoare făra să faci nimic!

```
return :: a -> IO a
```

Asemănatoar cu done, nu face nimic, dar produce o valoare.

Exemplu

```
return ""
```

- Dacă „șirul de intrare” conține "abc"
- atunci **return** "" produce:
  - valoarea ""
  - șirul (neschimbat) de intrare "abc"

Combinarea comenzilor cu valori

Operatorul de legare / bind

```
(>>=) :: 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

Mai multe detalii

```
(>>=) :: 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!

```haskell
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"

Operatorul de legare e similar cu **let**

Operatorul **let**

```
let x = m in n
```

let ca aplicație de funcții

```
(\ x -> n) m
```

Operatorul de legare

```
m >>= \ x -> n
```

De la intrare la ieșire

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

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

Test

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

Citirea unei linii în notație „do”

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

Echivalent cu:

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

Echo în notația „do”

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

Echivalent cu

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

Notația „do” în general

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

De exemplu

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

e echivalent cu

```
e1   >>= \x1 ->
e2   >>= \x2 ->
e3   >>
e4   >>= \x4 ->
e5   >>
e6
```

Clasa de tipuri **Monad**

```
class  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 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

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:

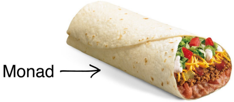
```
Prelude> :k Char
*     -- constructor de tip fara argumente
Prelude> :k  []
[] :: * -> *    -- constructor de tip cu un argument
```

Constructorii de tip pot fi și ei grupați în clase.

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

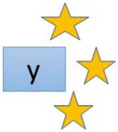
- Funcție îmbogățită: $x \mapsto$ 

  știind x, putem să extragem y și 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 modificarea unei stari

```
newtype State st a = State { runState :: st -> (a, st)}
rnd :: State Word32 Word32
rnd = State f
  where f seed = (seed', seed')
        seed' = cMULTIPLIER * seed + cINCREMENT
```

Exemple: bind (>>=)

- 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: bind (>>=)

- 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]
```

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

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
```

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

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

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

solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 0           -- a * x^2 + b * x + c = 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 listelor (a funcțiilor nedeterministe)

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

Rezultatul funcției e lista tuturor valorilor posibile.

```haskell
radical :: Float -> [Float]
radical x | x >= 0 = [negate (sqrt x), sqrt x]
          | x < 0  = []

solEq2 :: Float -> Float -> Float -> [Float]
solEq2 0 0 c = []                    -- a * x^2 + b * x + c = 0
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)
```

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
```

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

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)
```

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!"
```

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 >>=

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