Programare declarativă

Introducere în programarea funcțională folosind Haskell

Traian Florin Șerbănuță - seria 33
Ioana Leuștean - seria 34

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

1. Evaluare leneșă. Memoizare
2. Funcții de ordin înalt: foldr și foldl
3. Proprietatea de universalitate a funcției **foldr**
4. Generarea funcțiilor cu **foldr**

Evaluare leneșă. Memoizare

Funcțiile sunt valori

Funcțiile — „cetățeni de rangul I”

Funcțiile sunt valori!

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

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

```
Prelude> map ($ 3) [(4 +), (10 *), (^ 2), sqrt]
```

Funcțiile sunt valori

Funcțiile — „cetățeni de rangul I”

Funcțiile sunt valori!

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

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

```
Prelude> map ($ 3) [(4 +), (10 *), (^ 2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

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

```
filter p l = [x | x <- l, p x]
```

```
Prelude> filter (>= 2) [1,3,4]
[3,4]
```

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.

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.

Optimizarea recursiei

Memoizare

- 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*).

Optimizarea recursiei

Memoizare

- Să presupunem că vrem să optimizăm generarea șirului Fibonacci

```haskell
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

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:

```haskell
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?

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.

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:

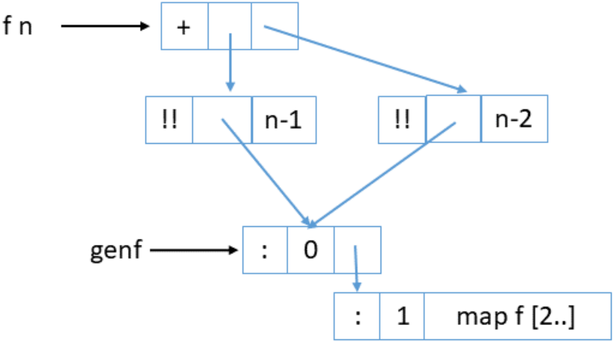
```
f :: Int -> Integer
f 0 = 0
f 1 = 1
f n = (genf !! (n-2)) + (genf !! (n-1))


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

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

Optimizarea recursiei: memoizare

Observați că elementele lui genf sunt evaluate o singură dată!



Optimizarea recursiei

- altă variantă, tot cu memoizare

```haskell
genf' :: Int -> Integer
genf'  =  l !! n
      where
       l = map f [0..]
       f 0 = 0
       f 1 = 1
       f n = (l !! (n-1)) + (l !! (n-2))
```

- o variantă simplă, fără memoizare, dar care depinde de forma particulară a recursiei:

```haskell
next (a : b : t) = (a + b) : next (b : t)
next _ = []  -- doar pentru a testa next pe liste finite

fibs = 0 : 1 : next fibs
```

Funcții de ordin înalt: foldr și foldl

foldr și foldl

Definiție

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.

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

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

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

```
Prelude> foldr (*) 1 [1,3,4]
12                    -- product [1,3,4]
```

Funcții de ordin înalt

foldr și foldl

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

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

```
Prelude> foldr (*) 1  [1,3,4]
12                        -- product [1,3,4]
```

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

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

```
Prelude> foldl (flip (:)) [] [1,3,4]
```

Funcții de ordin înalt

foldr și foldl

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

```
foldr o z [a1, a2, a3, ..., an] =
```

a1 ‘o‘ (a2 ‘o‘ (a3 ‘o‘ (... (an ‘o‘ z) ...)))

```
Prelude> foldr (*) 1  [1,3,4]
12                      -- product [1,3,4]
```

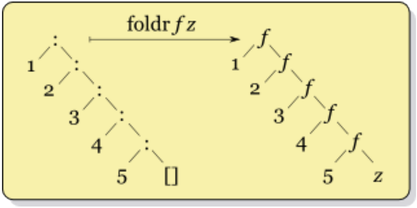
foldl :: (b -> a -> b) -> b -> [a] -> b

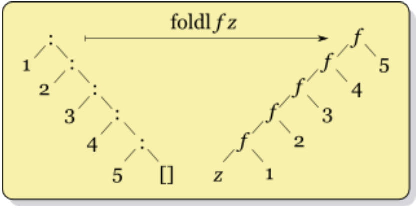
```
foldl o z [a1, a2, a3, ..., an] =
```

(... (((z ‘o‘ a1) ‘o‘ a2) ‘o‘ a3) ‘o‘ ... an)

```
Prelude> foldl (flip (:)) [] [1,3,4]
[4,3,1]   -- de ce? intelegeti modul de functionare!
```

foldr și foldl

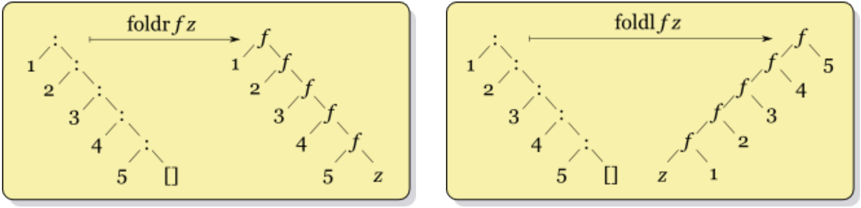




https://en.wikipedia.org/wiki/Fold_(higher-order_function)

Care dintre ele poate fi folosită pe liste infinite?

foldr și foldl



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

Care dintre ele poate fi folosită pe liste infinite?

- **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 evalueaza la infinit
```

Filtrare, transformare, agregare

Suma pătratelor elementelor pozitive

- Folosind descrieri de liste și funcții de agregare standard

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

- Folosind funcții auxiliare

```haskell
f xs = foldr (+) 0 (map sqr (filter pos xs))
  where
    sqr x = x * x
    pos x = x > 0
```

- Folosind funcții anonime

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

Filtrare, transformare, agregare

Suma pătratelor elementelor pozitive

- Folsind secțiuni și operatorul $ (parametru explicit)

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

- Definiție compozițională (pointfree style)

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

Proprietatea de universalitate a funcției **foldr**

Proprietatea de universalitate

Observație

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

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

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

Demonstrație:

$\Rightarrow$ Înlocuind $g = foldr\ f\ i$ se obține definiția lui **foldr**

$\Leftarrow$ Prin inducție dupa lungimea listei.

Proprietatea de universalitate

Observație

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

Teoremă

Fie g o funcție care procesează liste finite. Atunci

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

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

Generarea funcțiilor cu **foldr**

Compunerea funcțiilor

În signatura lui **foldr**

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

a si b pot fi de tip funcție.

Compunerea funcțiilor

În signatura lui **foldr**

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

a si b pot fi de tip funcție.

a si b sunt (c->c)

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

Compunerea funcțiilor

În signatura lui **foldr**

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

a si b pot fi de tip funcție.

a si b sunt (c->c)

```
compose :: [c -> c] -> (c -> c)
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
```

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)
```

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 $[x_1, \ldots, x_n]$ $0 = (\ldots(0 + x_1) + x_2) + \ldots x_n)$

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 $[x_1, \ldots, x_n]$ $0 = (\ldots(0 + x_1) + x_2) + \ldots x_n)$

- Observăm că

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

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

Definirea suml cu foldr

Proprietatea de universalitate

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

Definirea suml cu foldr

Proprietatea de universalitate

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

Gândim suml ca o funcție care are ca argument de tipul [**Int**] și întoarce o funcție de tipul **Int** -> **Int**.

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

Definirea suml cu foldr

Proprietatea de universalitate

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

Gândim suml ca o funcție care are ca argument de tipul [**Int**] și întoarce o funcție de tipul **Int** -> **Int**.

```
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)
```

$$
\begin{array}{lcll}
suml\ (x : xs) & = & f\ x\ (suml\ xs) & \text{egalitate de funcții} \\
suml\ (x : xs)\ n & = & f\ x\ (suml\ xs)\ n & \text{aplicăm funcțiile} \\
suml\ xs\ (n + x) & = & f\ x\ (suml\ xs)\ n &
\end{array}
$$

Definirea suml cu foldr

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

$$
\begin{array}{lcll}
suml\ (x : xs) & = & f\ x\ (suml\ xs) & \text{egalitate de funcții} \\
suml\ (x : xs)\ n & = & f\ x\ (suml\ xs)\ n & \text{aplicăm funcțiile} \\
suml\ xs\ (n + x) & = & f\ x\ (suml\ xs)\ n &
\end{array}
$$

Notăm $u = suml\ xs$ și obținem

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

Definirea suml cu foldr

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

$$\begin{array}{lcll} suml\ (x:xs) & = & f\ x\ (suml\ xs) & \text{egalitate de funcții} \\ suml\ (x:xs)\ n & = & f\ x\ (suml\ xs)\ n & \text{aplicăm funcțiile} \\ suml\ xs\ (n+x) & = & f\ x\ (suml\ xs)\ n & \end{array}$$

Notăm $u = suml\ xs$ și obținem

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

Rezultă că f = \x u n -> u (n+x), adică f = \x u -> (\n -> u (n+x))
deci f este o funcție cu două argumente care întoarce o funcție, al doilea argument fiind de asemenea o funcție.

Exemplu

```
Prelude> f= \ x u -> (\n -> u (x+n))
Prelude> f 3 (\x -> x+1) 5
9
```

Definirea sum cu foldr

Soluție

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

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

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

-- sum xs = suml xs 0
```

Definirea sum cu foldr

Soluție

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

```
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
```

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

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 = foldla h xs i
           where
             foldla h [] i = i
             foldla h (x:xs) i = foldl' h xs (h i x)
  -- foldla este o functie auxiliara
```

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 = foldla h xs i
            where
              foldla h [] i = i
              foldla h (x:xs) i = foldl' h xs (h i x)
  -- foldla este o functie auxiliara
```

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

(foldla h) cu **foldr**

foldla h :: [a] -> (b ->b)

funcție care are ca argument o lista si intoarce o funcție, deci g din proprietatea de universalitate este foldla h

Observăm că

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

(foldla h) cu **foldr**

foldla h :: [a] -> (b ->b)

funcție care are ca argument o lista si intoarce o funcție, deci g din proprietatea de universalitate este foldla h

Observăm că

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

Vrem să găsim *f* astfel încât

$$foldla\ h\ (x : xs) = f\ x\ (foldla\ h\ xs)$$

deoarece, din proprietatea de universalitate, va rezulta că

$$foldla\ h = foldr\ f\ id$$

foldl cu foldr

Soluție

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

foldla h = foldr f id

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


foldl h i xs = foldla 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
```

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 = (:[])

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

Prelude> take 3 (myfoldl (++) [] (map sing [1..]))
Interrupted.
-- myfoldl nu functioneaza pe liste infinite
```

Pe săptămâna viitoare!