Programare funcțională

Introducere în programarea funcțională folosind Haskell
C05

Ana Iova
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Foldr și Foldl

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 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)...)))
```

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

Funcția foldr

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

foldr și foldl

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
```

Compunerea funcțiilor

În definiția lui **foldr**

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

b poate fi tipul unei funcții.

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
```

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> compose [(+1), (^2)] 3
10
-- functia (foldr (.) id [(+1), (^2)]) aplicata lui 3
```

Suma elementelor dintr-o listă

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

Soluție cu **foldr**

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

Suma elementelor dintr-o listă

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:

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

Suma elementelor dintr-o listă

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:

$$\textbf{sum}\ [x_1, \ldots, x_n] = (x_1 + (x_2 + \ldots (x_n + 0) \ldots))$$

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

Suma elementelor dintr-o listă

sum cu acumulator

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

Suma elementelor dintr-o listă

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 elementelor dintr-o listă

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:

$$\text{suml } [x_1, \ldots, x_n]\ 0 = (\ldots(0 + x_1) + x_2) + \ldots x_n)$$

```
sum :: [Int] -> Int
sum xs = foldr (\ x u n -> u (n+x)) id xs 0
-- sum xs = suml xs 0
```

Inversarea elementelor unei liste

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

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]
```

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:

rev $[x_1, \ldots, x_n] = (\ldots(([] <:> x_1) <:> x_2).....) <:> x_n$

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:

rev $[x_1, \ldots, x_n] = (\ldots(([] <:> x_1) <:> x_2).....) <:> x_n$

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.

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.

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

Evaluare lenesă. 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]
```

Evaluare lenesă. 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 lenesă

- 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 lenesă. 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 lenesă

- 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 lenesă: lista numerelor prime

Vă amintiți din primul curs:

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

Evaluare lenesă: lista numerelor prime

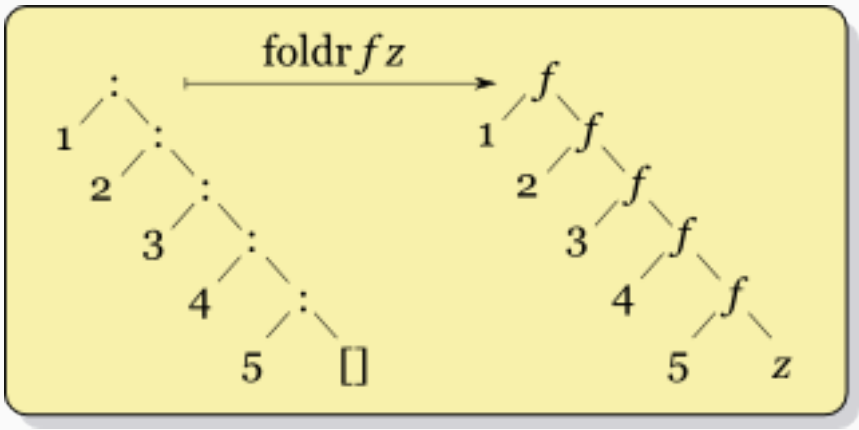
Vă amintiți din primul curs:

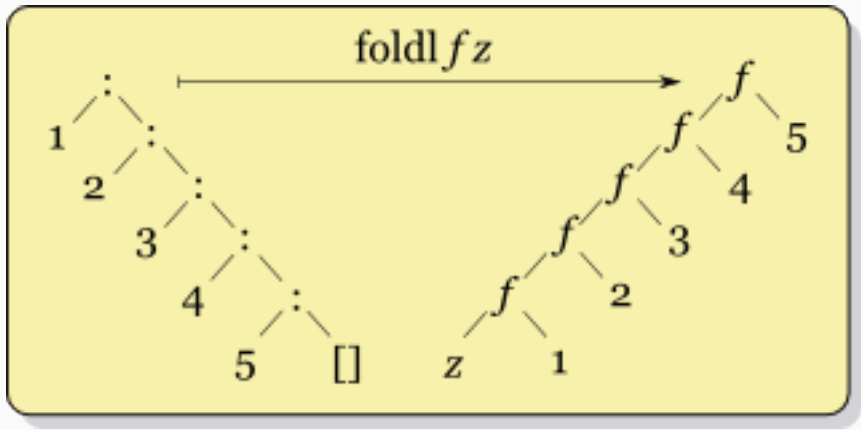
```
primes = sieve [2..]
sieve (p:ps) = p : sieve [ x | x <- ps, mod x p /= 0 ]
```

Intuitiv, evaluarea lenesă 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





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 calculeaza la infinit
```

Quiz time!

Seria 23: https://www.questionpro.com/t/AT4qgZphqg

Seria 24: https://www.questionpro.com/t/AT4NiZphmW

Seria 25: https://www.questionpro.com/t/AT4qgZphqv

Pe săptămâna viitoare!