

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C04

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Currying

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.

- În Haskell toate funcțiile sunt în forma **curry**, deci au un singur argument.
- Operatorul \rightarrow pe tipuri este asociativ la dreapta, adică tipul $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n$ îl gândim ca $a_1 \rightarrow (a_2 \rightarrow \dots (a_{n-1} \rightarrow a_n) \dots)$.
- Aplicarea funcțiilor este asociativă la stânga, adică expresia $f\ x_1 \dots x_n$ o gândim ca $(\dots ((f\ x_1)\ x_2) \dots x_n)$.

Funcțiile curry și uncurry și mulțimi

```
Prelude> :t curry
```

```
curry :: ((a, b) -> c) -> a -> b -> c
```

```
Prelude> :t uncurry
```

```
uncurry :: (a -> b -> c) -> (a, b) -> c
```

Exemplu:

```
f :: (Int , String) -> String
```

```
f (n,s) = take n s
```

```
Prelude> let cf = curry f
```

```
Prelude> :t cf
```

```
cf :: Int -> String -> String
```

```
Prelude> f(1,"abc")
```

```
"a"
```

```
Prelude> cf 1 "abc"
```

```
"a"
```

Funcții în matematică

- Fie $f : A \times B \rightarrow C$ 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 $f_x : B \rightarrow C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$. Funcția f_x se obține prin aplicarea parțială a funcției f .
- Dacă notăm $B \rightarrow C \stackrel{not}{=} \{h : B \rightarrow C \mid h \text{ funcție}\}$ observăm că $f_x \in B \rightarrow C$ pentru orice $x \in A$.
- Asociem lui f funcția $cf : A \rightarrow (B \rightarrow C)$, $cf(x) = f_x$.
- Vom spune că funcția cf este *forma curry* a funcției f .

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvHu0>

Seria 24: <https://questionpro.com/t/AT4NiZu3KB>

Seria 25: <https://questionpro.com/t/AT4qgZvHuP>

Agregarea elementelor dintr-o listă - fold

Exemplu - Suma

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

Soluție recursivă

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

```
Prelude> sum [1,2,3,4]
10
```


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

Soluție recursivă

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

```
Prelude> product [1,2,3,4]
```

24

Exemplu - Concatenare

Definiți o funcție care concatenează o listă de liste.

Soluție recursivă

```
concat :: [[a]] -> [a]
```

```
concat [] = []
```

```
concat (xs:xss) = xs ++ concat xss
```

```
Prelude> concat [[1,2,3],[4,5]]
```

```
[1,2,3,4,5]
```

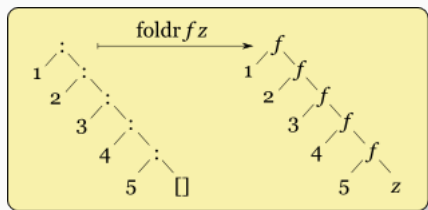
```
Prelude> concat ["con","ca","te","na","re"]
```

```
"concatenare"
```

Funcția foldr

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.



Funcția foldr

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

Soluție recursivă

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

foldr f i [] = i

foldr f i (x:xs) = f x (**foldr** f i xs)

Soluție recursivă cu operator infix

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

foldr op i [] = i

foldr op i (x:xs) = x `op` (**foldr** f i xs)

Exemplu — Suma

Soluție recursivă

sum :: [Int] -> Int

sum [] = 0

sum (x:xs) = x + **sum** xs

Soluție folosind foldr

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

sum :: [Int] -> Int

sum xs = **foldr** (+) 0 xs

Exemplu

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

Exemplu — Produs

Soluție recursivă

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

Soluție folosind foldr

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

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

Exemplu

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

Exemplu — Concatenare

Soluție recursivă

concat :: [[a]] -> [a]

concat [] = []

concat (xs:xss) = xs ++ **concat** xss

Soluție folosind foldr

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

concat :: [Int] -> Int

concat xs = **foldr** (++) [] xs

Exemplu

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

Exemplu – Suma pătratelor numerelor pozitive

```
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
```


Exemplu – Suma pătratelor numerelor pozitive

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

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

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

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

Exemplu - 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

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvALu>

Seria 24: <https://questionpro.com/t/AT4NiZvIWL>

Seria 25: <https://questionpro.com/t/AT4qgZvALw>

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 :: (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

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

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

Cu **foldr**, elementele sunt procesate de la dreapta la stânga:

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

Suma elementelor dintr-o listă

Soluție în care elementele sunt procesate de la stânga la dreapta.

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

Elementele sunt procesate de la stânga la dreapta:

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

Soluție cu foldl

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


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

```
--      flip  :: (a -> b -> c) -> (b -> a -> c)
--      (:)   :: a -> [a] -> [a]
--  flip  (:) :: [a] -> a -> [a]
```

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

Elementele sunt procesate de la stânga la dreapta:

rev $[x_1, \dots, x_n] = (\dots(([] \text{<:>} x_1) \text{<:>} x_2) \dots) \text{<:>} x_n$

Evaluare leneșă. Liste infinite

Putem folosi funcțiile **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..] -->
```

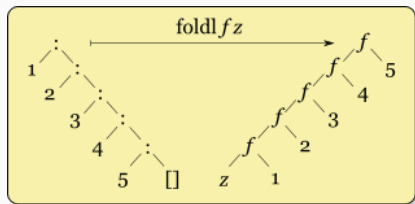
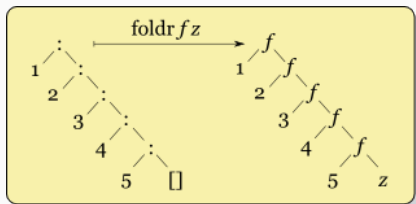
```
2 : sieve [ x | x <- [3..], mod x 2 /= 0 ] -->
```

```
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



[https://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](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://questionpro.com/t/AT4qgZvHue>

Seria 24: <https://questionpro.com/t/AT4NiZvIW4>

Seria 25: <https://questionpro.com/t/AT4qgZvHuf>

Pe săptămâna viitoare!