

Laboratorul 4: Funcția ‘foldr’, Evaluare Leneșă

RECOMANDARE Înainte de a începe să lucrați exercițiile din acest laborator finalizați exercițiile din laboratoarele precedente.

În acest laborator vom exersa conceptele prezentate în cursul 3.

I. Funcția foldr.

Funcția `foldr` este folosită pentru agregarea unei colecții. O definiție intuitivă a lui `foldr` este:

```
foldr op unit [a1, a2, a3, ... , an] == a1 `op` a2 `op` a3 `op` .. `op` an `op` unit
```

Vom exersa folosirea funcției `foldr` scriind câteva funcții, mai întâi folosind recursie, apoi folosind `foldr`.

Exercițiul 1

- (a) Scrieți o funcție recursivă care calculează produsul numerelor dintr-o listă.

```
produsRec :: [Integer] -> Integer
produsRec = undefined
```

- (b) Scrieți o funcție echivalentă care folosește `foldr` în locul recursiei.

```
produsFold :: [Integer] -> Integer
produsFold = undefined
```

Exercițiul 2

- (a) Scrieți o funcție recursivă care verifică faptul că toate elementele dintr-o listă sunt `True`.

```
andRec :: [Bool] -> Bool
andRec = undefined
```

- (b) Scrieți o funcție echivalentă care folosește `foldr` în locul recursiei.

```
andFold :: [Bool] -> Bool
andFold = undefined
```

Exercițiul 3

- (a) Scrieți o funcție recursivă care concatenează o listă de liste.

```
concatRec :: [[a]] -> [a]
concatRec = undefined
```

- (b) Scrieți o funcție echivalentă care folosește `foldr` în locul recursiei.

```
concatFold :: [[a]] -> [a]
concatFold = undefined
```

Exercițiul 4

- (a) Scrieți o funcție care elimină un caracter din șir de caractere.

```
rmChar :: Char -> String -> String
rmChar = undefined
```

- (b) Scrieți o funcție recursivă care elimină toate caracterele din al doilea argument care se găsesc în primul argument.

```
rmCharsRec :: String -> String -> String
rmCharsRec = undefined
```

```
test_rmchars :: Bool
test_rmchars = rmCharsRec ['a'..'l'] "fotbal" == "ot"
```

- (c) Scrieți o funcție echivalentă cu cea de la (b) care folosește `foldr` în locul recursiei.

```
rmCharsFold :: String -> String -> String
rmCharsFold = undefined
```

II. Evaluarea leneșă

Introducere

Haskell este un limbaj leneș. Asta înseamnă că:

1. Evaluarea unei expresii este amânată până când devine necesară pentru continuarea execuției programului. În particular, argumentele unei funcții nu sunt evaluate înainte de apelul funcției.
2. Chiar și atunci când devine necesară pentru continuarea execuției programului, evaluarea se face parțial, doar atât cât e necesar pentru a debloca execuția programului.
3. Pentru a evita evaluarea acelui argument al unei funcții de fiecare dată când e folosit în corpul funcției, toate aparițiile unei variabile sunt partajate, expandarea parțială a evaluării făcându-se pentru toate simultan.

Vom folosi în continuare o funcție intenționat definită ineficient pentru a testa ipotezele de mai sus. Funcția `logistic` simulează o lege de evoluție și a fost propusă ca generator de numere aleatoare.

```
logistic :: Num a => a -> a -> Natural -> a
logistic rate start = f
  where
    f 0 = start
    f n = rate * f (n - 1) * (1 - f (n - 1))
```

Pentru simplificare vom lucra cu o variantă a ei în care `rate` și `start` au fost instanțiate:

```
logistic0 :: Fractional a => Natural -> a
logistic0 = logistic 3.741 0.00079
```

Exercițiul 1

Pentru exercițiile de mai jos avem nevoie de o expresie a cărei execuție durează foarte mult timp, pentru a putea observa dacă este evaluată sau nu (și pentru a nu folosi `undefined`).

Testați că evaluarea funcției `logistic0` crește exponențial cu valoarea argumentului de intrare. Alegeți o valoare a acestuia `ex1` suficient de mare pentru a putea fi siguri dacă expresia se evaluează sau nu.

```
ex1 :: Natural
ex1 = undefined
```

Observație: chiar dacă nu rezolvați acest exercițiu, puteți observa dacă `logistic0 ex1` se evaluează deoarece `undefined` va arunca o excepție.

Amânarea evaluării expresiilor

Exercițiul 2

Evaluarea căroră dintre expresiile definite mai jos va necesita evaluarea expresiei `logistic0 ex1`?

Încercați să răspundeți singuri la întrebare, apoi testați în interpretor.

```
ex20 :: Fractional a => [a]
ex20 = [1, logistic0 ex1, 3]
```

```
ex21 :: Fractional a => a
ex21 = head ex20
```

```
ex22 :: Fractional a => a
ex22 = ex20 !! 2
```

```
ex23 :: Fractional a => [a]
ex23 = drop 2 ex20
```

```
ex24 :: Fractional a => [a]
ex24 = tail ex20
```

Evaluarea parțială a expresiilor

Exercițiul 3

Definim următoarele funcții auxiliare:

```
ex31 :: Natural -> Bool
ex31 x = x < 7 || logistic0 (ex1 + x) > 2
```

```
ex32 :: Natural -> Bool
ex32 x = logistic0 (ex1 + x) > 2 || x < 7
```

Evaluarea căroră dintre expresiile definite mai jos va necesita evaluarea expresiei `logistic0 (ex1 + x)`?

Încercați să răspundeți singuri la întrebare, apoi testați în interpretor.

```
ex33 :: Bool
ex33 = ex31 5
```

```
ex34 :: Bool
ex34 = ex31 7
```

```
ex35 :: Bool
ex35 = ex32 5
```

```
ex36 :: Bool
ex36 = ex32 7
```