Laboratorul 5: Evaluare leneșă, proprietatea de universalitate a funcției ‘foldr‘, matrici

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

I. 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. In 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 aceluiaș argument al unei funcții de fiecare dată cănd 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.

```haskell
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 cărora dintre expresiile definite mai jos va necesita evaluarea expresiei `logistic0 ex1`?

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

```haskell
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:

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

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

Evaluarea cărora 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
```

II. Universalitatea funcției `foldr`

O posibilă definiție a funcției `foldr` ar putea fi cam așa:

```
foldr_ :: (a -> b -> b) -> b -> ([a] -> b)
foldr_ op unit = f
  where
    f []     = unit
    f (a:as) = a `op` f as
```

Această definiție ne dă și o indicație despre ce funcții recursive pe liste pot fi definite folosind `foldr` și cum putem să derivăm aceste definiții, astfel:

Dată fiind o funcție `f :: [a] -> b` pentru care putem descoperi `unit :: b` și `op :: a -> b -> b` astfel încât `f [] = unit` și `f (a:as) = op a (f as)`, atunci avem că `f = foldr op unit`.

Exemplul 1: Suma pătratelor elementelor impare

```
sumaPatrateImpare :: [Integer] -> Integer
sumaPatrateImpare []     = 0
sumaPatrateImpare (a:as)
  | odd a = a * a + sumaPatrateImpare as
  | otherwise = sumaPatrateImpare as
```

Aplicând algoritmul de mai sus, putem defini varianta ei folosind `foldr` în locul recursiei:

```
sumaPatrateImpareFold :: [Integer] -> Integer
sumaPatrateImpareFold = foldr op unit
  where
    unit = 0
```

```
    a `op` suma
      | odd a     = a * a + suma
      | otherwise = suma
```

Exemplul 2: funcția `map`

```
map_ :: (a -> b) -> [a] -> [b]
map_ f []     = []
map_ f (a:as) = f a : map_ f as
```

Aplicăm algoritmul de mai sus pentru a obține `map_ f`:

```
mapFold :: (a -> b) -> [a] -> [b]
mapFold f = foldr op unit
  where
    unit = []
    a `op` l = f a : l
```

Exemplul 3: funcția `filter`

```
filter_ :: (a -> Bool) -> [a] -> [a]
filter_ p [] = []
filter_ p (a:as)
  | p a       = a : filter_ p as
  | otherwise = filter_ p as
```

Aplicăm algoritmul de mai sus pentru a obține `filter_ p`:

```
filterFold :: (a -> Bool) -> [a] -> [a]
filterFold p = foldr op unit
  where
    unit = []
    a `op` filtered
      | p a       = a : filtered
      | otherwise = filtered
```

Exercițiul 1

(a) Folosind doar recursie și funcții de bază, scrieți o funcție `semn` care ia ca argument o listă de întregi și întoarce un șir de caractere care conține semnul numerelor din intervalul `-9..9` (inclusiv), ignorându-le pe celelalte.

Indicație: `String = [Char]`

```
semn :: [Integer] -> String
semn = undefined
```

```
test_semn :: Bool
test_semn = semn [5, 10, -5, 0] == "+-0" -- 10 este ignorat
```

(c) Folosiți algoritmul descris mai sus pentru a defini funcția semn folosind `foldr` în locul recursiei

```
semnFold :: [Integer] -> String
semnFold = foldr op unit
  where
    unit = undefined
    op = undefined
```

III. Matrici

Matrice ca liste. Vom considera matricele ca fiind liste de liste, de exemplu

```
matrice :: Num a => [[a]]
matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Exercițiul 1

Vom lucra cu tipuri generale, astfel că, în general, o matrice va avea signatura `[[a]]`. Pentru a ne asigura că matricile diferite sunt corecte, vom defini predicatul:

```
corect :: [[a]] -> Bool
corect = undefined
```

care are scopul de a verifica faptul că toate listele au aceeași lungime. De exemplu,

```
-- corect [[1, 2, 3], [4, 5, 6], [7, 8, 9]] == True
```

va întoarce `True`, în timp ce

```
-- corect [[1, 2, 3], [4, 5, 6], [7, 9]] == True
```

va întoarce `False`.

Exercițiul 2

Vrem să definim un predicat pentru a obține elementul de la o anumită poziție din matrice, bazat pe linia și coloana acestuia, considerând indexarea de la 0. Pentru matricea de la exercițiul anterior, apelul `el [[1,2,3],[4,5,6],[7,8,9]] 0 1` va afișa 2.

```
el :: [[a]] -> Int -> Int -> a
el = undefined
```

Exercițiul 3

În acest exercițiu, vrem să obținem o listă în care fiecare element să aibă asociată poziția din matrice (linia și coloana pe care se află situat).

```
transforma :: [[a]] -> [(a, Int, Int)]
transforma = undefined
```