Haskell: Evaluare leneșă, programare point-free și list comprehensions

- Data publicării: 28.03.2022
- Data ultimei modificări: 28.03.2022

Obiective

Scopul acestui laborator îl reprezintă acomodarea cu noțiuni avansate de programare în Haskell, având în vedere scrierea de cod clar și concis.

Aspectele urmărite sunt:

- list comprehensions
- definirea de liste infinite
- programare "point-free"

List comprehensions

Haskell oferă un mod suplimentar de a genera liste: scriem proprietățile pe care ar trebui să le respecte elementele listei într-o sintaxă numită **list comprehension**. Este o sintaxă similară celei din matematică. De exemplu, vrem lista numerelor pare, divizibile cu 3. În matematică, am fi avut ceva de tipul $\{x \mid x \in N2, x \equiv 0 \pmod 3\}$ (pentru N2 mulțimea numerelor pare). În Haskell, avem

```
> [x | x <- [0, 2 ..], x `mod` 3 == 0] -- lista numerelor naturale pare, divizibile cu 3
[0,6,12,18,24,30,36,42,48,54,60,66,72,78,84,90,96,102,108, Interrupted.
```

Observați că se generează elemente la infinit. Pentru fiecare element din lista `[0, 2 ..]` (din expresia `x <- [0, 2..]`) se testează condițiile următoare. Dacă toate sunt îndeplinite, se generează elementul din fața `|`.

Pentru a putea vedea o porțiune a fluxului folosim functiile `take` și `drop`.

Liste infinite și generarea lor folosind funcționale

Haskell facilitează generarea de liste infinite, dată fiind natura **leneșă** a evaluării expresiilor. De exemplu mulțimea numerelor naturale poate fi definită după cum urmează:

```
naturals = [0..]
```

Definiția de mai sus este însă **zahăr sintactic** pentru următoarea expresie (exemplificată anterior):

```
naturals = iter 0
    where iter x = x : iter (x + 1)
```

Putem de asemenea să generăm liste infinite folosind funcționala iterate [http://hackage.haskell.org/package/base-4.6.0.1/docs/Prelude.html#v:iterate], având prototipul:

```
> :t iterate
iterate :: (a -> a) -> a -> [a]
```

`iterate` primește o funcție `f` și o valoare inițială `x` și generează o listă infinită din aplicarea repetată a lui `f`. Implementarea listei numerelor naturale va arăta deci astfel:

```
naturals = iterate (\x -> x + 1) 0  -- SAU
naturals = iterate (+ 1) 0
```

Observăm că `iterate` este nu numai o funcțională, ci și un **șablon de proiectare** (design pattern). Alte funcționale cu care putem genera liste infinite sunt:

- `zipWith`: generalizare a lui `zip` care, aplică o funcție binară pe elementele a două liste; e completată de `zipWith3` (pentru funcții ternare), `zipWith4` etc.
- `foldl`, `foldr`, `map`, `filter`

De asemenea, există funcții, care nu primesc ca parametru alte funcții, cu care putem genera liste infinite, de exemplu:

- `repeat`: repetă o valoare la infinit
- `intersperse`: introduce o valoare între elementele unei liste

Exemple de utilizare:

```
ones = repeat 1 -- [1, 1, 1, ..]
onesTwos = intersperse 2 ones -- [1, 2, 1, 2, ..]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs) -- sirul lui Fibonacci
powsOfTwo = iterate (* 2) 1 -- puterile lui 2
palindromes = filter isPalindrome [0..] -- palindroame
    where
    isPalindrome x = show x == reverse (show x) -- truc: reprezint numarul ca String
```

Point-free programming

În Haskell, compunerea funcțiilor se realizează cu ajutorul operatorului `.`. De interes este și operatorul `$` definit ca:

```
f $ x = f x
```

El are avantajul de a grupa expresiile din dreapta și stânga lui înainte de aplicarea funcției, scăpând astfel de paranteze:

```
length $ 3 : [1, 2] -- length (3 : [1, 2])
```

Împreună cu `curry`, `uncurry` și `flip` aceste funcții duc la un stil de programare în care punctul (de exemplu, `x`) în care se evaluează funcția nu este explicitat. Urmăriți exemplul următor de transformare:

```
square x = x*x
inc x = x+1
f1 x = inc (square x)
f2 x = inc $ square x
f3 x = inc . square $ x
f4 = inc . square
```

Stilul are câteva avantaje în domeniul expresivității și al verificării programului dar, folosit excesiv, poate duce la cod obscur, greu de înțeles.

"Point-free style" [https://wiki.haskell.org/Pointfree] reprezintă un stil de programare în care evităm menționarea explicită a parametrilor unei funcții în definiția acesteia. Cu alte cuvinte, se referă la scrierea unei funcții ca o succesiune de compuneri de funcții. Această abordare ne ajută, atunci când scriem sau citim cod, să ne concentrăm asupra obiectivului urmărit de algoritm deoarece expunem mai transparent

ordinea în care sunt efectuate operațiile. În multe situații, codul realizat astfel este mai compact și mai ușor de urmărit.

De exemplu, putem să definim operația de însumare a elementelor unei liste in felul următor:

```
sum xs = foldl (+) 0 xs
```

Alternativ, în stilul "point-free", vom evita descrierea explicită a argumentului funcției:

```
sum = foldl (+) 0
```

Practic, ne-am folosit de faptul că în Haskell toate funcțiile sunt în formă curry [https://wiki.haskell.org/Currying], aplicând parțial funcția `foldl` pe primele doua argumente, pentru a obține o expresie care așteaptă o listă și produce rezultatul dorit.

Pentru a compune două funcții, vom folosi operatorul `.`:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Dacă analizăm tipul operatorului, observăm că acesta primește ca argumente o funcție care acceptă o intrare de tipul `b` și întoarce o valoare de tipul `c`, respectiv încă o funcție care primește o intrare de tipul `a` și produce o valoare de tipul `b`, compatibilă cu intrarea așteptată de prima funcție. Rezultatul întors este o funcție care acceptă valori de tipul `a` și produce rezultate de tipul `c`.

Cu alte cuvinte, expresia `(f . g)(x)` este echivalentă cu `f(g(x))`.

Spre exemplu, pentru a calcula expresia `2 * x + 1` pentru orice element dintr-o listă, fără a folosi o funcție lambda, putem scrie direct:

```
> let sm = map ((+ 1) . (* 2))
> :t sm
sm :: [Integer] -> [Integer]
```

De observat că au fost necesare paranteze pentru ca întreaga expresie `(+ 1) . (* 2)` să reprezinte un singur argument pentru funcția `map`. Am putea obține un cod mai concis folosind operatorul `$`:

```
> let sm = map $ (+ 1) . (* 2)
> :t ($)
($) :: (a -> b) -> a -> b
```

`$` este un operator care aplică o funcție unară pe parametrul său. Semantica expresiei `f $ x` este aceeași cu cea a `f x`, diferența fiind pur sintactică: precedența `$` impune ordinea de grupare astfel încât expresia din stânga `$` este aplicată pe cea din dreapta. Avantajul practic este că putem sa omitem parantezele în anumite situații, în general atunci când ar trebui să plasăm o paranteză de la punctul unde se află `$` până la sfârșitul expresiei curente.

De exemplu, expresiile următoare sunt echivalente:

```
> length (tail (zip [1,2,3,4] ("abc" ++ "d")))
> length $ tail $ zip [1,2,3,4] $ "abc" ++ "d"
```

Alternativ, folosind operatorul de compunere `.`:

```
> (length . tail . zip [1,2,3,4]) ("abc" ++ "d")
> length . tail . zip [1,2,3,4] $ "abc" ++ "d"
```

Atenție!: Cei doi operatori, `.` și `$` nu sunt, în general, interschimbabili:

```
> let f = (+ 1)
> let g = (* 2)
> :t f . g
f . g :: Integer -> Integer
> :t f $ g
-- eroare, f așteaptă un Integer, g este o funcție
> f . g $ 2
5
> f . g 2
-- eroare, echivalent cu f . (g 2), f . g (2)
> f $ g $ 2
5
> f $ g 2
5
```

Uneori, ne dorim să schimbăm ordinea de aplicare a argumentelor pentru o funcție. Un exemplu de funcție care ne-ar putea ajuta în acest sens este funcția `flip`, care interschimbă parametrii unei funcții binare:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

De exemplu, dacă vrem să construim o funcție point-free care aplică o funcție primită ca argument pe fluxul numerelor naturale, am putea scrie:

```
> :t map
map :: (a -> b) -> [a] -> [b]
> :t flip map
flip map :: [a] -> (a -> b) -> [b]
> let f = flip map $ [0..]
> take 10 $ f (*2)
[0,2,4,6,8,10,12,14,16,18]
```

Un șablon de proiectare des folosit este `concatMap`, sau map/reduce [https://en.wikipedia.org/wiki/MapReduce], care implică compunerea funcționalelor `map` și `fold`. De exemplu, funcția `intersperse`, prezentată anterior, poate fi definită sub forma:

```
myIntersperse :: a -> [a] -> [a]
myIntersperse y = foldr (++) [] . map (: [y])
```

Observăm că al doilea argument al funcției `myIntersperse` nu este menționat explicit.

Atenție! Folosirea stilului "point-free" poate să scadă în anumite cazuri lizibilitatea codului! Ideal este să folosiți construcții "point-free" doar atunci când acestea fac programul **mai ușor** de înțeles. Proiectele Haskell de dimensiuni mari încurajează stilul "point-free" **doar** împreună cu clauze `let` și `where`, și uneori cu funcții anonime.

Un exemplu de aplicație uzuală de "point-free style programming", cu care probabil sunteți deja familiari, este folosirea operatorului pipe ("|") în unix shell scripting [http://www.vex.net/~trebla/weblog/pointfree.html].

Resurse

- Cheatsheet [https://github.com/cs-pub-ro/PP-laboratoare/raw/master/haskell/intro/haskell-cheatsheet-1.pdf]
- Schelet [https://ocw.cs.pub.ro/courses/_media/pp/22/laboratoare/haskell/evaluare-lenesa-schelet.zip]
- Soluții [https://ocw.cs.pub.ro/courses/_media/pp/22/laboratoare/haskell/evaluare-lenesa-solutii.zip]

Referințe

- *Pointfree on Haskell Wiki [http://www.haskell.org/haskellwiki/Pointfree]*
- *Pointfree Style - Whay is it good for [http://buffered.io/posts/point-free-style-what-is-it-good-for]*

- *Advantages of point-free on Lambda the Ultimate [http://lambda-the-ultimate.org/node/3233]*
- *Haskell: function composition (.) vs function application ($) (SO) [http://stackoverflow.com/questions/3030675/haskell-function-composition-and-function-application-idioms-correct-us]*
- *Pointfree (Haskell Wiki) [http://www.haskell.org/haskellwiki/Pointfree]*
- *Higher order function (Haskell Wiki) [http://www.haskell.org/haskellwiki/Higher_order_function]*