Programare Funcţională în Haskell

Alan Perlis zicea că *un limbaj care nu afectează modul în care gândim nu merită cunoscut*. Judecând după numărul mare de întrebări pe Stack Overflow corelat cu numărul impresionant de articole, publicații și postări pe reddit, Slashdot și bloguri, este evident că limbajul Haskell este un limbaj ce merită cunoscut. De fapt, programarea funcțională în sine reprezintă un domeniu pe care fiecare programator ar trebui să-l cunoască puțin.

Istoria a făcut ca limbajele de programare răspândite acum să fie cele din paradigma imperativă. Codul scris de un programator nu este altceva decât un set de instrucțiuni și ordine date calculatorului. Dar foarte puțini știu că paradigma imperativă nu reprezintă decât unul dintre cele patru modele de calcul inventate de matematicienii ce au pus bazele științei calculatoarelor. Foarte diferite, aceste paradigme au totuși un lucru în comun: conform tezei Church-Turing nici o paradigmă nu poate rezolva mai multe probleme decât alta. Cu toate acestea unele programe sunt mai ușor de scris într-o manieră imperativă în timp ce altele sunt mai ușor de descris într-un limbaj declarativ. Acest lucru este vizibil în însăși preferința programatorilor de azi: deși toate paradigmele au apărut în aproximativ același an, faptul că este mult mai simplu de făcut un calculator pe modelul von Neumann a făcut ca cea imperativă să domine lumea limbajelor de programare mult timp. Cu toate acestea, celelalte și-au găsit și ele rolul, în special în inteligența artificială. Cel mai vechi limbaj de programare folosit și acum – LISP – este de altfel meritul lui John McCarthy și al inteligenței artificiale.

Un moment important este dat de publicarea articolului influențial al lui John Bachus *Poate fi programarea eliberată de stilul von Neumann? Stilul funcțional și o algebră a programelor*. Acesta aduce la lumină un punct important: programele funcționale se pot compune mult mai ușor din asamblarea unor componente testate anterior. Practic, acest articol a produs apariția unor limbaje funcționale noi pe lângă LISP. Astfel, în anii 1970 au apărut nu mai puțin de 10 limbaje de programare funcționale noi. Dintre acestea, limbajele din categoria ML reprezintă un interes major prin două limbaje: OCaml folosit și în prezent și Miranda.

De la aceste limbaje au pornit în anii 1990 și programatorii adunați într-un comitet creat special pentru inventarea unui limbaj de programare ce ar trebui văzut ca un standard deschis pentru cercetarea în domeniul limbajelor de programare (nu doar funcționale) și al compilatoarelor: Haskell.

Cu timpul Haskell a reușit să ajungă un limbaj de programare puternic, destul de folosit în domenii diverse precum inteligența artificială, securitate, dezvoltarea aplicațiilor web sau a sistemelor de operare. Multe din noutățile ce au fost introduse în limbaj au ajuns mai târziu și în C++, Java, etc.

Această dezvoltare se datoarează atât comunității create în jurul limbajului (canalul de IRC #haskell de pe Freenode, grup special pe Reddit și Stack Overflow, o mulțime de bloguri colectate săptămânal de *Haskell Weekly News* și bianual de *Haskell Communities and Activities Report*) cât și facilității cu care poate fi învățat limbajul folosind resurse online (*The Monad Reader*, wiki-uri și wikibooks, etc.).

Nume celebre din industria IT folosesc în prezent Haskell sau în producție sau pentru dezvoltare locală: NASA, Galois, Facebook, Google, Microsoft, Jane Street, etc. Primul interpretor pentru Perl6 a fost scris integral în Haskell. Instrumente pentru validarea automată a programelor sau demonstrarea de teoreme sunt dezvoltate în Haskell (uneori împreună cu Ocaml).

Dar ce are limbajul atât de atrăgător? Majoritatea celor care îl folosesc se vor lega de două

aspecte: stilul declarativ și garanția unei anumite corectitudini a codului. Programele scrise în limbaje funcționale sunt extrem de concise, se pune accentul pe ce ar trebui să facă secevența de cod, nu pe cum ar trebui să facă asta. De exemplu, secvența următoare sortează o listă de elemente cu un algoritm de tip quick-sort (se alege mereu drept pivot primul element din lista sortată):

```
sort [] = []
sort (x:xs) = lesser ++ x:greater
  where
    lesser = sort [a | a <- xs, a < x]
    greater = sort [a | a <- xs, a >= x]
```

Comparați lungimea acestui cod cu cea a unui program similar scris în limbajul de programare favorit. În mod cert, stilul funcțional beneficiază de aspectul declarativ: codul este mai ușor de citit, elemente necunoscute de sintaxă pot fi deduse din context, corectitudinea este ușor de validat.

Pentru a veni în ajutorul validării corectitudinii programului, Haskell are tipare statică: fiecare expresie are un tip cunoscut de la compilare. În plus, nu există conversii implicite între tipuri similare: programatorul va trebui să facă explicit conversiile în locurile în care acestea sunt necesare. Unii cercetători au promovat ideea că o folosire corectă a tipurilor poate duce la garantarea unui invariant important: dacă programul compilează atunci sunt șanse mari ca el să fie corect. De cele mai multe ori acest lucru nu este adevărat în totalitate. De fapt, se încearcă transformarea celor mai frecvente erori de runtime și bug-uri în erori de compilare. De exemplu, limbajul Haskell rezolvă problema null-pointer-exception folosind un tip de date special numit Maybe: programatorul va întoarce valori încapsulate în acest tip din funcțiile care pot eșua și compilatorul se asigură că orice folosire a rezultatului întors de aceste funcții este verificată. La extrem, s-au creat framework-uri de dezvoltare de aplicații Web în care fiecare șir de caractere are un tip propriu: nu se mai pot confunda porțiuni de CSS cu porțiuni de HTML și nici nu se mai pot face atacuri clasice de tip SQL Injection sau similare.

Un alt avantaj al tipării statice este faptul că un programator poate căuta o funcție cunoscând doar tipul acesteia. Există două motoare de căutare pentru acest lucru: Hayoo și Hoogle. Ambele vor căuta funcții încercând să generalizeze cât mai mult posibil argumentele oferite și să ofere sugestii particulare cât mai elocvente. După un timp, se ajunge la a programa ca și cum s-ar rezolva un puzzle uriaș în care piesele sunt date de funcțiile ce vor ajunge în programul final.

O calitate importantă a limbajului o reprezintă puritatea. Ținând cont că orice program trebuie să interacționeze cu exteriorul (citire date, transmitere date, etc), Haskell oferă și el funcții pentru operațiile de intrare-ieșire. Dar, acestea vor fi marcate diferit din punct de vedere al tipurilor. Tipul oricărei acțiuni cu efecte laterale este IO. Dacă din Maybe se puteau scoate valorile încapsulate, acest lucru nu mai este posibil în cazul IO. Asta face ca orice funcție ce folosește efecte laterale să nu poată fi chemată decât din funcții ce vor fi marcate ca având efecte laterale. Practic, codul este împărțit în 2 tabere: codul pur funcțional pentru prelucrarea de date și codul imperativ pentru interacțiunea cu exteriorul. Această separare face ca Haskell să fie considerat de Simon Peyton Jones – unul dintre creatorii limbajului – ca fiind cel mai bun limbaj de programare *imperativ*.

Segregarea efectelor laterale și a funcțiilor pure asigură paralelizarea ușoară a codului. Este extrem de simplu pentru un programator să paralelizeze codul scris în Haskell: neavând efecte laterale funcțiile pure pot fi apelate în orice ordine și de oricâte ori, rezultatele pot fi memoizate în cache-uri, se poate scrie cod fără a fi necesare primitivele de sincronizare.

Un alt atu important al limbajului este evaluarea leneșă. Programatorul poate scrie cod ce

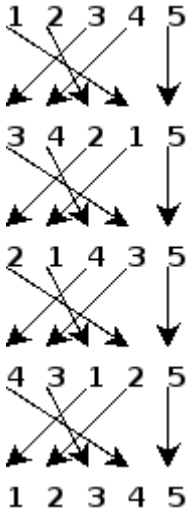
lucrează pe date infinite cât timp acestea au o reprezentare finită și este necesară doar evaluarea unui număr finit al acestora. Gândiți-vă la un șir de la matematică: este o structură infinită dar poate fi reprezentat finit printr-o recurență și când îi scrieți termenii vă opriți mereu după primii 3-4.

Deși evaluarea leneșă permite scrierea de programe circulare și/sau lucrând cu secvențe infinite, uneori aceasta deranjează la analiza programului din punct de vedere al perfomanței. Din fericire, limbajul oferă construcții pentru controlul evaluării fiind astfel posibil ca programatorul să declare că anumiți membri sau anumite argumente ale unei funcții să fie evaluați imediat. Pe de altă parte, evaluarea leneșă asigură că se evaluează exact cât este necesar din datele cerute, irosind cât mai puțini cicli de ceas.

De fapt, îmbinând toate facilitățile limbajului se poate scrie cod declarativ/funcțional având aceeași performanță cu limbajele clasice C, Java, etc. Avantajul folosirii unui limbaj funcțional este evident dacă vă gândiți că un cod declarativ este ușor de citit și la 2-3 luni după scrierea acestuia: e mult mai facilă corectarea bug-urilor în momentul în care acestea apar.

Codul Haskell poate fi interpretat sau compilat. Se recomandă folosirea suitei GHC (*The Glorious Glasgow Haskell Compiler Collection*) împreună cu *Haskell Platform* pentru a avea toate beneficiile limbajului și ale unor biblioteci de nivel înalt foarte performante și testate. Suita GHC vine cu un compilator (ghc) și un interpretor (ghci). Interpretorul permite dezvoltarea rapidă a aplicațiilor, testarea codului sau a tipului unor expresii, etc. Compilatorul permite generarea unui executabil având aceleași performanțe cu ale unui cod scris în limbajele clasice. Suita *Haskell Platform* oferă utilitare de profiling a aplicației pentru a optimiza codul cât mai mult posibil.

Considerând că am prezentat destule detalii despre limbaj pentru a provoca pasiunea de a-l învăța, este timpul să trecem la noțiuni de sintaxă. La final, în acest articol se va scrie un program ce va rezolva o problemă simplă: dându-se o permutare să se afle ordinul acesteia (numărul de aplicări ale permutării asupra permutării identitate pentru a se ajunge înapoi la permutarea identitate). De exemplu, pentru permutarea (3 4 2 1 5), răspunsul căutat este 4: aplicând (3 4 2 1 5) peste (1 2 3 4 5) vom ajunge la (3 4 2 1 5), apoi la (2 1 4 3 5), (4 3 1 2 5) și înapoi la (1 2 3 4 5) (aplicarea unei permutări presupune mutarea elementului de pe poziția i pe poziția corespunzătoare din permutare).



Vom reprezenta o permutare folosind o listă a numerelor de la 1 la n. În Haskell, [1, 2, 3] reprezintă lista primelor 3 numere. Pentru a avea un cod generic, putem instanția lista folosind un

generator, ca în [1 .. n]. O listă din Haskell permite adăugarea de elemente doar în față. Pentru aceasta se folosește operatorul :. Lista vidă este reprezentată de []. Elementele unei liste sunt toate de același tip. Pentru a accesa un element, putem folosi operatorul !!.

Majoritatea funcțiilor în programarea funcțională sunt recursive. Astfel, se poate demonstra corectitudinea codului folosind inducție structurală. De exemplu, funcția următoare va calcula lungimea unei liste:

```
length [] = 0
length (x:xs) = 1 + length xs
```

Prima linie spune că lungimea listei vide este 0. A doua spune că lungimea unei liste formate prin inserarea unui element x în fața unei lista xs este cu 1 mai mare decât lungimea listei xs.

Programatorii obișnuiți cu părțile low-level și structura compilatoarelor vor spune că recursivitatea consumă spațiu pe stivă și că nu este un pattern care ar trebui folosit tot timpul. Pe de altă parte, compilatorul din suita GHC poate realiza transformări asupra funcției astfel încât să se ajungă la a folosi recursivitatea de tip coadă: fiecare apel recursiv înlocuiește cadrul de stivă al apelului anterior astfel încât rezultatul este întors direct în funcția apelantă.

Dacă vă uitați la cele 2 linii pentru funcția length, veți vedea că există câte una pentru fiecare constructor al listei. Practic, majoritatea funcțiilor în programarea funcțională pornesc de la deconstruirea tipului de date: se scrie câte o expresie pentru fiecare constructor al tipului în parte.

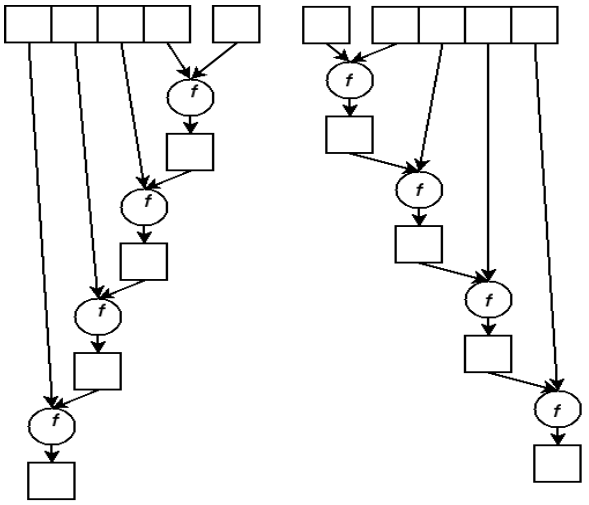
Considerați acum funcțiile următoare. Prima calculează suma elementelor dintr-o listă în timp ce a doua calculează produsul lor.

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

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

După cum observați, există un șablon comun urmărit de toate funcțiile: există un element inițial pentru cazul în care pornim de la lista vidă și o operație pe care trebuie să o aplicăm peste elementul curent și rezultatul apelului recursiv. Practic, având aceste 2 ingrediente putem “împături” lista într-o nouă valoare. Acest șablon este capturat de Haskell prin intermediul a două funcții (diferența fiind dată de direcția în care se realizează reducerea):

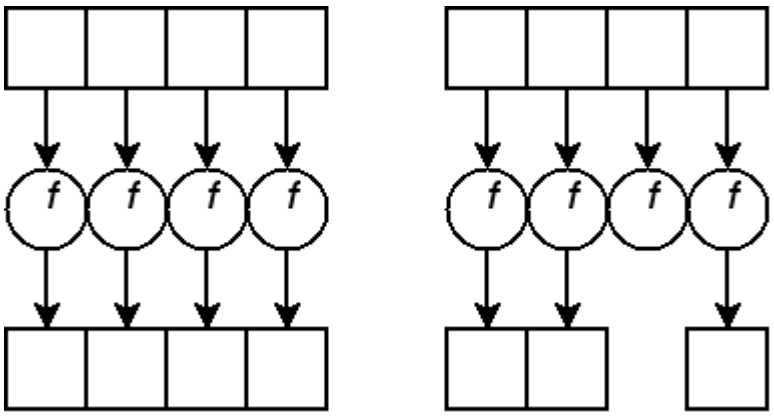
```
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)

foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs
```

Observați că una dintre cele 2 funcții este tail-recursive de la început: apelul recursiv se efectuează cu unul dintre argumente actualizat cu rezultatul parțial de până atunci. Diferența între cele 2 poate fi observată și din imaginea următoare.



Alte două pattern-uri importante capturate de Haskell sunt map și filter. Le vom ilustra direct prin imaginea următoare și codul următor:

```
map f [] = []
map f (x:xs) = f x : map f xs

filter p [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs
```



Practic, folosirea șabloanelor și funcțiilor predefinite ajută pentru a scrie mai puțin cod (amintiți-vă principiul DRY) și pentru a ajunge la un cod pentru care este mai ușor de demonstrat corectitudinea. În plus, compilatorul poate avea optimizări suplimentare incorporate pentru aceste funcții speciale.

Putem scrie acum cod pentru a aplica o permutare asupra unei liste: pentru fiecare element din permutare întoarcem elementul de pe poziția respectivă din listă (folosind !! pentru a obține elementul și map pentru a efectua operația pentru toate elementele).

```
applyPerm :: [Int] -> [Int] -> [Int]
applyPerm l p = map (\i -> l !! (i - 1)) p
```

Având această funcție, se poate calcula răspunsul problemei folosind funcția until. Aceasta este oarecum echivalenta unei bucle while din programarea imperativă: primește un predicat (funcție ce întoarce True/False), o funcție pentru a itera de la o stare la următoarea și starea inițială și va întoarce starea finală, în momentul în care predicatul devine True.

```
computeOrder :: [Int] -> Int
computeOrder perm = fst $ until test f $ f init
  where
    l = length perm - 1
    test p = snd p == [1 .. l]
    f (a, list) = (a+1, applyPerm list perm)
    init = (0, [1 .. l])
```

Cum funcționează funcția? Pornind de la o permutare perm, vom aplica applyPerm iterativ pentru a modifica starea curentă din until. Pentru stare reținem două valori într-un tuplu: un contor și lista curentă, rezultatul permutărilor.

Putem testa codul scris de noi în interpretorul ghci:

```
*Main> computeOrder [3,4,2,1,5]
4
```

Rezultatul este 4, exact cum ne așteptam. Putem testa să vedem dacă rezultatul este corect. Folosim iterate pentru a apela o funcție la infinit, generând lista [x, f x, f (f x), ...]. Folosim take pentru a lua doar primele elemente dintr-o listă. Astfel, profităm de evaluarea leneșă pentru a calcula doar elementele necesare.

```
*Main> take 5 $ iterate (flip applyPerm [3, 4, 2, 1, 5]) [1..5]
[[1,2,3,4,5],[3,4,2,1,5],[2,1,4,3,5],[4,3,1,2,5],[1,2,3,4,5]]
```

În ediția viitoare vom prezenta mai multe elemente legate de sintaxa Haskell: modul prin care se pot defini tipuri noi și modul în care acestea pot fi folosite. În mod cert, aplicația din articolul următor va fi mai complexă decât cea de acum.