Racket: Întârzierea evaluării

- Responsabil: Mihaela Balint [mailto:oornot@gmail.com]

- Data publicării: 16.03.2019
- Data ultimei modificări: 16.03.2019

Obiective

Scopul acestui laborator este înțelegerea diverselor tipuri de evaluare, respectiv a controlului evaluării în Racket.

Conceptele introduse sunt:

- evaluare aplicativă
- evaluare leneșă
- promisiuni
- fluxuri

Evaluare aplicativă vs. evaluare leneșă

Fie următoarea aplicație, scrisă în **calcul Lambda**:

```
(λx.λy.(x + y) 1 2)
```

În urma aplicării funcției de mai sus, expresia se va evalua la `3`. Ce se întâmplă însă dacă parametrii funcției noastre reprezintă alte aplicații de funcții?

```
(λx.λy.(x + y) 1 (λz.(z + 2) 3))
```

Intuitiv, expresia de mai sus va aduna `1` cu `5`, unde `5` este rezultatul evaluării expresiei `(λz.(z + 2) 3)`. În cadrul acestui raționament, am presupus că parametrii sunt evaluați **înaintea** aplicării funcției asupra acestora. Vom vedea, în cele ce urmează, că evaluarea se poate realiza și în altă ordine.

Evaluare aplicativă

În **evaluarea aplicativă** (*eager evaluation*), o expresie este evaluată **imediat** ce o variabilă se leagă la aceasta. Pe exemplul de mai sus, **evaluarea aplicativă** decurge astfel:

```
(λx.λy.(x + y) 1 (λz.(z + 2) 3))
(λx.λy.(x + y) 1 5)
6
```

Observații:

- parametrii funcției sunt evaluați **înaintea** aplicării funcției asupra acestora; afirmăm că transferul parametrilor se face **prin valoare**;
- **`Racket`** (ca și majoritatea limbajelor tradiționale) folosește **evaluare aplicativă**.

Evaluare leneșă

Spre deosebire de **evaluarea aplicativă**, **evaluarea leneșă** va întârzia evaluarea unei expresii până când valoarea ei este necesară. Exemplu:

```
(λx.λy.(x + y) 1 (λz.(z + 2) 3))
(1 + (λz.(z + 2) 3))
```

```
(1 + 5)
6
```

Observații:

- aplicația funcției anonime de parametru `z` este trimisă ca parametru și nu se evaluează înainte ca acest lucru să devină necesar; afirmăm că transferul parametrilor se face **prin nume**;
- **`Haskell`** folosește **evaluare leneșă**.

Există avantaje și dezavantaje ale **evaluării leneșe**. O situație în care **evaluarea leneșă** se poate dovedi utilă este următoarea:

Fie funcția `Fix`:

```
Fix = λf.(f (Fix f))
```

`Fix` primește ca parametru o funcție `f`, care este aplicată asupra parametrului `(Fix f)`. Să considerăm funcția constantă (care întoarce mereu valoarea `b`):

```
ct = λx.b
```

Să aplicăm `Fix` asupra funcției constante `ct`, folosind **evaluarea aplicativă**:

```
(Fix ct)
(λf.(f (Fix f)) ct)
(ct (Fix ct))
(ct (λf.(f (Fix f)) ct))
(ct (ct (Fix ct)))
(ct (ct (λf.(f (Fix f)) ct)))
(ct (ct (ct (Fix ct))))
...
```

Cum evaluarea este **aplicativă**, parametrul trimis lui `ct`, mai precis `(Fix ct)`, va fi evaluat, în mod recursiv, la infinit.

Să încercăm aceeași aplicare, folosind de data aceasta **evaluarea leneșă**:

```
(Fix ct)
(λf.(f (Fix f)) ct)
(ct (Fix ct))
(λx.b (Fix ct))
b
```

Observăm că evaluarea **leneșă** a lui `(Fix ct)` se termină și întoarce valoarea `b`. Terminarea este posibilă datorită faptului că evaluarea expresiei `(Fix ct)`, trimisă ca parametru lui `ct`, este întârziată până când este nevoie de ea (în acest caz, niciodată).

Construcții precum `Fix` pot părea ezoterice și nefolositoare. În realitate însă, ele au aplicabilitate. `Fix` este un *combinator de punct fix* („generator“ de funcții recursive).

Evaluarea în Racket

Fie următorul cod:

```
(+ 1 (+ 2 3))
```

Codul de mai sus este o rescriere din **calcul Lambda** în **Racket** a exemplului introductiv. Reamintim că evaluarea în Racket este **aplicativă**, așadar etapele parcurse sunt următoarele:

- `(+ 2 3)`: al doilea parametru al funcției `+` se evaluează la `5`;
- `(+ 1 5)` se evaluează la `6`.

Pentru a obține beneficiile evaluării **leneșe** în Racket, putem **întârzia** evaluarea unei expresii în două moduri:

- inchideri funcționale (de obicei nulare (fără parametri)): `(lambda () (…))`;
- promisiuni: `delay`/`force`.

Evaluare leneșă folosind închideri

Fie următorul exemplu:

```
(define sum
  (lambda (x y)
    (lambda ()
      (+ x y))))
(sum 1 2)
```

Observați că `(sum 1 2)` nu evaluează suma parametrilor, ci întoarce o **funcție**. Mai precis, avem de-a face cu o **închidere funcțională** (funcție care își salvează contextul). Pentru a **forța** evaluarea este suficient să aplicăm rezultatul întors de `(sum 1 2)` asupra celor zero parametri pe care îi așteaptă, astfel:

```
((sum 1 2)) ; se va evalua la 3
```

Evaluare leneșă cu promisiuni

Fie definiția convențională a sumei între două numere:

```
(define sum
  (lambda (x y)
    (+ x y)))
```

Putem **întârzia** evaluarea unei sume astfel:

```
(delay (sum 1 2)) ; va afișa #<promise>
```

Pentru a scrie o funcție **cu evaluare leneșă**, asemănătoare celei scrise cu `(lambda () …)`, procedăm astfel:

```
(define sum
  (lambda (x y)
    (delay (+ x y))))

(sum 1 2) ; va afișa #<promise>
```

Pentru a forța evaluarea, folosim `force`:

```
(force (sum 1 2)) ; va afișa 3
```

Fluxuri - aplicații ale evaluării leneșe

Folosind evaluarea leneșă, putem construi **obiecte infinite** sau **fluxuri** (*streams*). Exemplu de *flux*: șirul numerelor naturale: `(0 1 2 3 … n … )`. Un astfel de obiect se reprezintă ca o **pereche** între:

- un **element curent** (primul element din flux, asemănător unui `car` pe liste);
- un **generator** (o promisiune sau o închidere funcțională) care, pornit (prin `force` sau aplicație), va întoarce următoarea pereche din *flux* (restul fluxului, asemănător unui `cdr` pe liste).

Exemple:

Șirul constant de 1

Fie șirul infinit:

```
a(n) = 1, n >= 0
```

Primele 5 elemente ale acestui șir sunt:

```
(1 1 1 1 1)
```

Șirul se poate genera astfel, folosind închideri funcționale:

```
(define ones-stream
  (cons 1 (lambda () ones-stream)))
```

Observăm că șirul este o pereche în care:

- primul element este **valoarea curentă**, `1`;
- al doilea element este o **funcție** ((lambda () ones-stream)) capabilă să genereze restul fluxului.

Folosind promisiuni, același șir este generat ca mai jos:

```
(define ones-stream
  (cons 1 (delay ones-stream)))
```

Limbajul Racket pune la dispoziție o interfață pentru manipularea fluxurilor, asemănătoare cu aceea pentru manipularea listelor:

- `stream-cons` este omologul lui `cons` (adaugă un element la începutul unui flux);
- `stream-first` este omologul lui `car` (extrage primul element din flux);
- `stream-rest` este omologul lui `cdr` (reprezintă fluxul fără primul său element);
- `empty-stream` este omologul lui `'()` (fluxul vid);
- `stream-empty?` este omologul lui `null?` (testează dacă un flux este vid);
- `stream-map`, `stream-filter` corespund lui `map`, `filter` (însă `stream-map` acceptă doar funcții unare).

În spiritul abstractizării, putem folosi această interfață fără să ne preocupe dacă funcțiile sunt implementate folosind închideri funcționale sau promisiuni (pentru curioși - sunt implementate folosind promisiuni). Șirul infinit de `1` se va genera astfel:

```
(define ones-stream
  (stream-cons 1 ones-stream))
```

Să scriem o funcție care întoarce primele `n` elemente din șir:

```
; extragerea primelor n elemente din șirul s
(define (stream-take s n)
  (cond ((zero? n) '())
        ((stream-empty? s) '())
        (else (cons (stream-first s)
                    (stream-take (stream-rest s) (- n 1))))))

; testare
(stream-take ones-stream 5) ; va afișa (1 1 1 1 1)
```

Șirul numerelor naturale

Fie următorul cod, care implementează șirul numerelor naturale:

```
; generator pentru numere naturale
(define (make-naturals k)
  (stream-cons k (make-naturals (add1 k))))

; fluxul numerelor naturale
(define naturals-stream (make-naturals 0))

; testare
(stream-take naturals-stream 4) ; va afisa (0 1 2 3)
```

Observații:

- în plus față de cum am generat șirul infinit de 1, aici am folosit o funcție, `add1`, care calculează elementul următor din flux pe baza elementului curent - din acest motiv, am definit un generator care primește ca parametru elementul curent;
- în funcție de fluxul construit, generatorul poate primi alți parametri.

Șirul lui Fibonacci

Șirul lui Fibonacci este:

```
Fibo = t0 t1 t2 t3 ... tn ...
```

```
unde:
t0 = 0
t1 = 1
tk = t(k-2) + t(k-1) pentru k >= 2
```

În cazul acestui șir, un nou element este calculat pe baza unor **valori** calculate **anterior**. Observăm că:

```
Fibo        = t0 t1 t2 t3 ... t(k-2) ... +
(tail Fibo) = t1 t2 t3 t4 ... t(k-1) ...
___________________________________________
Fibo  = t0 t1 t2 t3 t4 t5 ... tk ...
```

Adunând elemente din șirurile (fluxurile) `Fibo` și `(tail Fibo)`, obținem șirul `t2 t3 … tk …`. Dacă adăugăm la începutul acestui rezultat elementele `t0` și `t1`, obținem exact `Fibo`.

Pentru a implementa expresia de mai sus, avem un singur **obstacol** de depășit, și anume trebuie să scriem o funcție care **adună două fluxuri**. O implementare posibilă este:

```
(define add
  (lambda (s1 s2)
    (stream-cons (+ (stream-first s1) (stream-first s2))
                 (add (stream-rest s1) (stream-rest s2)))))
```

Definim fluxul `Fibo` pe baza adunării de mai sus:

```
(define fibo-stream
  (stream-cons 0
               (stream-cons 1
                            (add fibo-stream (stream-rest fibo-stream)))))
```

Fluxul numerelor prime

Eratostene a conceput un algoritm pentru determinarea fluxului numerelor prime, care funcționează astfel: fie șirul numerelor naturale, începând cu `2`:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 ...
_
```

Păstrăm primul element din șir (`2`) și eliminăm elementele care se divid cu el:

```
2 3   5   7   9    11    13    15    17    19    21 ...
_
```

Apoi păstrăm următorul element din șirul rămas (`3`) și eliminăm toate elementele care se divid cu el:

```
2 3   5   7        11    13          17    19       ...
  _
```

etc.

Algoritmul este implementat mai jos:

```
(define sieve
  (lambda (s)
    (let ((divisor (stream-first s)))
      (stream-cons
       ;; păstrează primul element
       divisor
       ;; și aplică recursiv algoritmul pe numerele care nu se divid cu el
       (sieve (stream-filter (lambda (x) (> (modulo x divisor) 0))
                             (stream-rest s)))))))

(define primes-stream
  (sieve
   ;; șirul numerelor naturale începând de la 2
   (stream-rest (stream-rest naturals-stream))))

; testare
(stream-take primes-stream 10) ; va afișa (2 3 5 7 11 13 17 19 23 29)
```

Resurse

- Exerciţii
- Cheatsheet
- Soluții

Referințe

- *Structure and Interpretation of Computer Programs [https://web.mit.edu/alexmv/6.037/sicp.pdf]*, ediția a doua
- *Evaluation strategy [http://en.wikipedia.org/wiki/Evaluation_strategy]*