Închideri funcţionale. Evaluare aplicativă şi evaluare întârziată

May 14, 2014

Obiective

- Evaluarea
  - aplicativă versus întârziată
  - în RACKET
- Evaluarea întârziată în RACKET
  - cu închideri funcţionale
  - cu promisiuni
- Aplicaţii: fluxuri (stream-uri)
  - Şirul constant 1
  - Numere naturale
  - Şirul Fibonacci
  - Şirul numerelor prime

Argumentele funcţiilor sunt evaluate înaintea aplicării func tiei asupra lor.

Example

$$\begin{aligned}(\lambda x.\lambda y.(x+y)\ 1\ \underline{(\lambda z.(z+2)\ 3)}) &= (\underline{\lambda x.\lambda y.(x+y)\ 1}\ 5)\\ &= \underline{\lambda y.(1+y)\ 5}\\ &= 6.\end{aligned}$$

Argumentele funcţiilor sunt evaluate înaintea aplicării func tiei asupra lor.

Example

$$\begin{aligned}(\lambda x.\lambda y.(x+y)\ 1\ \underline{(\lambda z.(z+2)\ 3)}) &= (\underline{\lambda x.\lambda y.(x+y)\ 1}\ 5)\\ &= \underline{\lambda y.(1+y)\ 5}\\ &= 6.\end{aligned}$$

Observaţii

- În această evaluare toate funcţiile la apel primesc valori ca şi argumente (transfer prin valoare).
- RACKET şi toate limbajele imperative folosesc mecanismul de evaluarea aplicativă.

Evaluarea leneşă a unei expresii va întârzia evaluarea până în momentul când aceasta este folosită efectiv.

Example

$$\underline{(\lambda x.\lambda y.(x+y)\ 1}\ (\lambda z.(z+2)\ 3)) = \underline{(\lambda y.(1+y)\ (\lambda z.(z+2)\ 3))} =$$
$$1 + \underline{(\lambda z.(z+2)}\ 3) = 1 + 5 = 6.$$

Evaluarea leneşă a unei expresii va întârzia evaluarea până în momentul când aceasta este folosită efectiv.

Example

$$\underline{(\lambda x.\lambda y.(x+y)\ 1}\ (\lambda z.(z+2)\ 3)) = \underline{(\lambda y.(1+y)\ (\lambda z.(z+2)\ 3))} =$$
$$1 + \underline{(\lambda z.(z+2)}\ 3) = 1 + 5 = 6.$$

Observaţii

- Apelul funcţiei $(\lambda z.(z+2)\ 3)$ este transmis ca parametru şi nu se evaluează înainte ca acest lucru să devină necesar.

Evaluarea aplicativă versus evaluarea leneşă
Avantaje, dezavantaje?

Amândouă au avantaje şi dezavantaje.

- Dezavantaje ale evaluării aplicative
  - ▷ Poate efectua calcule nefolositoare
  - ▷ Poate rula la infinit
- Dezavantaje ale evaluării leneşe:
  - ▷ Unele calcule pot fi duplicate

Example

Fie $fix = \lambda f.(f\ (fix\ f))$ şi $ct = \lambda x.1$.

- Evaluarea aplicativă a *fix ct* rulează la infinit:

$$\underline{fix\ ct} = ct\ (\underline{fix\ ct}) = ct\ (ct\ (\underline{fix\ ct})) = \ldots$$

- Evaluarea leneşă *fix ct* se opreşte imediat:

$$\underline{fix\ ct} = \underline{ct\ (fix\ ct)} = 1$$

Evaluarea leneşă în RACKET

Deşi RACKET foloseşte evaluarea aplicativă, putem simula evaluarea leneşă în două moduri:

1. Utilizând închideri funcţionale (funcţii nulare)
   - O închidere funcţională nulară este o expresie care conţine o expresie `lambda` cu zero argumente:

     `(lambda ()` *body*`)`
2. utilizând promisiuni `delay`/`force`

Evaluarea întârziată utilizând închideri funcţionale nulare

Ideea principală:

- Se defineşte o funcţie cu evaluare întârziată a cărui corp conţine `(lambda () ...)`

Example (Evaluarea întârziată pentru suma)

```
> (define suma
    (lambda (x y)
          (lambda ()
             (+ x y))))
> (suma 1 2)
#<procedure>
>((suma 1 2))
3
```

OBSERVAŢIE: `(sum 1 2)` returnează o funcţie cu 0 argumente (care e o închidere funcţională nulară).
Apelul unei functii nulare `f` este `(f)`.

Evaluarea leneşă utilizând `delay/force`

Example (Evaluarea întârziată pentru suma)

Definiţia pentru `suma` este:

```
> (define suma-1 (lambda (x y) (+ x y)))
```

Pentru a întârzia evaluarea, apelăm

```
> (define s (delay (suma-1 1 2)))
> s
#promise
```

Pentru a forţa evaluarea *c*, apelăm `(force` *c*`)`:

```
> (force s)
3
```

Evaluarea leneşă utilizând `delay/force`

Example (versiunea cu delay pentru suma)

```
> (define suma-2
    (lambda (x y)
      (delay (+ x y))))
> (suma-2 1 2)
#promise
> (force (suma-2 1 2))
3
```

Flux (Stream): o reprezentare finită a unei liste infinite.
Exemple de liste infinite care pot fi reprezentate ca stream-uri:

Şirul constant 1: `(1 1 1 ...)`

Şirul numerelor naturale: `(0 1 2 3 ...)`

Şirul Fibonacci: `(1 1 2 3 5 8 13 ...)`

$(f_0 = f_1 = 1\ f_{n+2} = f_n + f_{n+1}$ for all $n \geq 0)$

Şirul numerelor prime: `(2 3 5 7 11 13 ...)`

Ideea principală a reprezentării stream-urilor

$(a_1 \ \ldots \ a_k \ . \ \mathit{gen})$

unde*gen* este un generator folosit pentru a genera restul elementelor stream-ului. De regulă, *gen* este ori o funcţie nulară ori o promisiune.

Implementarea utilizând o funcţie nulară ca generator

```
> (define make-ones (lambda () (cons 1 make-ones)))
> (define all-ones (make-ones))
> ; extrage lista primelor n elemente ale stream-ului
   (define (take n stream)
     (if (= n 0)
         null
        (if (pair? stream)
            (cons (car stream)
                  (take (- n 1) (cdr stream)))
            (take n (stream)))))
> (take 4 all-ones) ; test
'(1 1 1 1)
```

Implementare utilizând promisiuni

```
; generator pentru numere naturale
(define make-naturals
   (lambda (k)
     (cons k (delay (make-naturals (+ k 1))))))
; sirul numerelor naturale incepand cu 0
(define all-naturals (make-naturals 0))
; extragerea primelor n elemente
(define (take n stream)
  (if (= n 0)
      null
      (if (pair? stream)
          (cons (car stream)
                (take (- n 1) (cdr stream)))
          (take n (force stream)))))
```

Observaţii

- Exerciţiu: Reimplementaţi `make-ones` utilizând `delay/force`, şi `all-naturals` utilizând funcţii nulare.

Exemplu: `take` care funcţionează pentru ambele tipuri de implementări

```
(define (take n stream)
  (if (= n 0)
      null
      (if (pair? stream)
          (cons (car stream)
                (take (- n 1) (cdr stream)))
          (take n (if (promise? stream)
                      (force stream)
                      (stream)))))))
```

Observaţie: şirul `fib` Fibonacci are următoarele proprietăţi:

- `fib` + `(cdr fib)` este `(cdr (cdr fib))`

$$\begin{array}{rcccccc} \texttt{fib} & = & f_0 & f_1 & f_2 & \ldots & + \\ \texttt{(cdr fib)} & = & f_1 & f_2 & f_3 & \ldots & \\ \hline f_0 \; f_1 & & f_2 & f_3 & f_4 & \ldots & \end{array}$$

Observaţie: şirul `fib` Fibonacci are următoarele proprietăţi:

- `fib` + `(cdr fib)` este `(cdr (cdr fib))`

$$\begin{array}{rcccccc} \texttt{fib} & = & f_0 & f_1 & f_2 & \ldots & + \\ (\texttt{cdr fib}) & = & f_1 & f_2 & f_3 & \ldots & \\ \hline f_0 \;\; f_1 & & f_2 & f_3 & f_4 & \ldots & \end{array}$$

- Odată ce cunoaştem primele 2 elemente f_0 şi f_1, putem începe să generăm restul şirului:

  `fib = (1  1 .` *gen*`)`

  unde generatorul *gen* implementează funcţia de adunare a `fib` cu `(cdr fib)`.

```
(define fib
   (cons 1 (cons 1 (delay (add fib (cdr fib))))))
; adunarea a doua stream-uri
(define (add s1 s2)
   (if (promise? s1)
      (add (force s1) s2)
      (if (promise? s2)
         (add s1 (force s2))
         (cons (+ (car s1) (car s2))
               (delay (add (cdr s1) (cdr s2)))))))
```

Test:

```
> (take 10 fib)
'(1 1 2 3 5 8 13 21 34 55)
```

Idea principală: ciurul lui Eratostene

- Se consideră şirul tuturor numerelor naturale care încep cu 2:
  `2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...`

Idea principală: ciurul lui Eratostene

- Se consideră şirul tuturor numerelor naturale care încep cu 2:
  `2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...`
- Se şterg toate elementele divizible cu primul element din şir, exceptând primul element

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

Idea principală: ciurul lui Eratostene

- Se consideră şirul tuturor numerelor naturale care încep cu 2:

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

- Se şterg toate elementele divizible cu primul element din şir, exceptând primul element

  2 3 5 7 9 11 13 15 17 19 ...

- Procesul se repetă pe coada a ceea ce rămâne:

  2 **3** 5 7 11 13 17 19 ...
  2 3 **5** 7 11 13 17 19 ...

  etc.

Applicaţii ale evaluării leneşe
Operaţii utile: filtrare şi mapare pe stream-uri

- `(myfilter` *pred stream*`)` returnează şirul elementelor din *stream* pentru care *pred* are loc.
- `(mymap` *f stream*`)` aplică funcţia unară *f* la toate elementele din *stream*

```
(define myfilter (lambda (p s)
   (if (promise? s)
      (myfilter p (force s))
      (if (p (car s))
         (cons (car s) (delay (myfilter p (cdr s))))
         (myfilter p (cdr s))))))

(define mymap (lambda (f s)
   (if (promise? s)
      (mymap f (force s))
      (cons (f (car s)) (delay (mymap f (cdr s)))))))
```

```
; auxiliare
(define (divides? x y) (zero? (remainder x y)))
; ciurul lui Eratostene
(define (sieve s)
  (if (promise? s)
      (sieve (force s))
      (cons (car s)
            (delay (sieve
               (myfilter
                  (lambda (x) (not (divides? x (car s))))
                  (cdr s)))))))
(define all-primes (sieve (make-naturals 2)))
```

Tests

```
> ; primele 10 numere prime
  (take 10 all-primes)
'(2 3 5 7 11 13 17 19 23 29)
> ; primele numere prima mai mari decˆ at 200
  (car (myfilter (lambda (x) (> x 200)) all-primes))
211
> ; primele 10 multiple de 5
   (take 10
      (mymap (lambda (x) (* 5 x)) all-naturals))
'(0 5 10 15 20 25 30 35 40 45)
> ; primele 10 elemente din fib + all-primes
   (take 10 (add fib all-primes))
'(3 4 7 10 16 21 30 40 57 84)
```