Legarea variabilelor. Închideri funcţionale

May 7, 2014

Obiective

- scopul (sau domeniul de vizibilitate) a unei variabile
- legarea statică şi legarea dinamică
- context computational
- închidere funcţională
- întârzierea evaluării (folosind închideri funcţionale)

Domeniu de vizibilitate

- Ce este o variabilă?

Domeniu de vizibilitate

- Ce este o variabilă?
  - O pereche identificator-valoare.

Domeniu de vizibilitate

- Ce este o variabilă?
  - O pereche identificator-valoare.
- Domeniu de vizibilitate al unei variabile este mulţimea punctelor din program în care valoarea poate fi accesată prin referirea identificatorului.

Domeniu de vizibilitate

- Ce este o variabilă?
  - ▷ O pereche identificator-valoare.
- Domeniu de vizibilitate al unei variabile este mulţimea punctelor din program în care valoarea poate fi accesată prin referirea identificatorului.
- In limbajele de programare, o variabilă este:

  Legată static (lexical scoping): domeniul de vizibilitate al variabilelor este controlat textual, prin mecanisme specifice limbajului.

  Legată dinamic (dynamic scoping): domeniul de vizibilitate al unei variabile este controlat dinamic, în funcţie de timp

Domeniu de vizibilitate

- Ce este o variabilă?
  - ▷ O pereche identificator-valoare.
- Domeniu de vizibilitate al unei variabile este mulţimea punctelor din program în care valoarea poate fi accesată prin referirea identificatorului.
- In limbajele de programare, o variabilă este:

  Legată static (lexical scoping): domeniul de vizibilitate al variabilelor este controlat textual, prin mecanisme specifice limbajului.

  Legată dinamic (dynamic scoping): domeniul de vizibilitate al unei variabile este controlat dinamic, în funcţie de timp
- In RACKET, unele variabile sunt legate static, în timp ce altele sunt legate dinamic.

Legarea statică

`lambda`, `let`, `let*`

Legarea statică se face utilizând următoarele:

- `(lambda` *args* *body*`)`
- `(let ([`x_1 e_1`] ... [`x_n e_n`])` *body*`)`
  se leagă $\{x_1 : v_1, \ldots, x_n : v_n\}$
- `(let* ([`x_1 e_1`] ... [`x_n e_n`])` *body*`)`
- se leagă $x_1 : v_i$ (unde v_i este valoarea lui e_i) este $e_{i+1}, \ldots, e_n$ şi *body*

Legarea statică

- Construcţia let permite crearea de variabile ce vor fi vizibile în corpul let-ului. Sintaxa este:

```
(let ((<id1> <val1>)    ; id1 se leaga la val1
      (<id2> <val2>)    ; etc
      ...
      (<idn> <valn>))
   <expr1>
   <expr2>
   ...
   <exprn>)
```

Legarea statică

- Codul anterior este echivalent cu:

```
((lambda (<id1> ...<idn>)
    <expr1>
    ...
    <exprn>)
 <val1>
 ...
 <valn>)
```

- `let` şi `let*` sunt abrevieri (ca de exemplu *syntactic sugar*).

Legarea statică

- Indicaţi valoarea returnată în urma evaluării:

```
(define a 10)
(let ((a 1) (b (+ a 1)))
          ; aici suntem in zona de
          ; definitii, nu in corpul let-ului
          ; => a e legat la 10
  (cons a b)) ; în corpul let-ului este
              ; vizibila legarea lui a la 1
```

- Indicaţi valoarea returnată în urma evaluării următorului program:

```
#lang racket
(let ((a 1))                ; prima legare
  (let ((f (lambda () a)))
    (let ((a 2))            ; a doua legare
      (f))))    ; afiseaza 1
```

Legarea statică (let*)

- Este asemanator cu let, insa domeniul de vizibilitate al variabilelor începe imediat dupa definire.
- Indicaţi valoarea returnată în urma evaluării:

```
(define a 10)
(let* ((a 1) (b (+ a 1)))
            ; în momentul definirii lui b,
            ; este vizbila legarea lui a la 1
  (cons a b)) ; desigur, aceeasi legare e
            ; vizibila si în corpul let-ului
```

(letrec ([x_1 e_1] ... [x_n e_n]) *body*)

- Domeniul de vizibilitate al fiecarui x_i este e_1,...,e_n, *body*.
  - Valoarea v_i a lui e_1 este asignata la x_i si este accesibila doar in e_{i+1},...,e_n, si *body*.
  - In e_1, ..., e_{i-1}, variabila x_i are valoarea `#<undefined>`

Example

```
> (letrec ([a b] [b 1])
    (cons a b))
'(#<undefined> . 1)
   ; în momentul definirii lui a este nevoie
    ; de valoarea lui b, necunoscuta înca
    ; de aceea a se leaga la #<undefined>
```

Un exemplu: Definirea a doua functii dependente una de alta

```
> ((letrec
    ((even-length?
      (lambda (L)    ; even-length? este o
                     ;  închidere functionala
        (if (null? L)        ; deci corpul functiei
                             ; nu este evaluat la
            #t               ; momentul definirii ei
            (odd-length? (cdr L))))) ;deci nu e o proble
                 ; ca înca nu stim cine e odd-length?
     (odd-length?
      (lambda (L)
        (if (null? L)
            #f
            (even-length? (cdr L))))))
  (even-length? '(1 2 3 4 5 6))) ; in acest moment deja
                                 ; ambele functii au fost definite
#t
```

Named `let`

Aceasta constructie se foloseste pentru a obtine un mod de a itera in interiorul unei functii. Numele dat let-ului (în cazul de mai jos, iter) refera la o procedura care primeste ca parametri variabilele definite de let si care evalueaza expresiile din corpul let-ului. Exemplul urmator tipareste pe cate o linie numerele de la 1 la 3:

Example

```
> (define count
    (lambda (n)
      (let iter  ([i 0])
        (display i)
        (newline)
        (if (< i n) (iter (+ i 1)) i))))
> (count 3)
1
2
3
3
```

Legarea dinamică

Variabilele introduse global utilizand `define` sunt "dynamically scoped":

- ▷ legarea lor se poate modifica printr-un nou `define`
- ⇒ domeniul de vizibilitate al unei variabile este controlat dinamic, in functie de timp.

Example

```
> (define a 1)
> (define f (lambda (x) (+ x a)))
> (f 2)           ; a e legata dinamic la 1
3
> (define a 2)    ; a e legata dinamic la 2
> (f 2)
4
```

- Un dezavantaj al legarii dinamice este pierderea transparentei referentiale (acelasi apel - (f 2) in exemplul nostru - intoarce doua valori diferite).

Închideri funcţionale

- Definim contextul computational al unui punct P din program la un moment t ca fiind multimea variabilelor vizibile in punctul P la momentul t. Mai exact, contextul computational al unui punct P din program contine perechi de tipul (identificator valoare). Valoarea asociata unui identificator poate varia in timp exclusiv in cazul variabilelor definite cu define - dupa cum am observat in exemplul anterior.
- Conceptul de inchidere functionala a fost inventat în anii ’60 si implementat pentru prima data in Scheme. Pentru a intelege acest concept, sa ne gandim ce se intampla in Racket cand definim o functie, de exemplu functia (define f (lambda (x) (+ x a)))
- Ceea ce face orice define este sa creeze o pereche identificator-valoare; in acest caz se leaga identificatorul f la valoarea produsa de evaluarea lambda-expresiei (lambda (x) (+ x a))

Închideri funcţionale

O inchidere functionala este:

- textul lambda-expresiei ((lambda (x) (+ x a)) pe exemplul nostru)
- contextul computational in punctul de definire a lambda-expresiei ((a 1) pe exemplul nostru)
- Ceea ce salvam în context este de fapt multimea variabilelor libere în lambda-expresia noastra, adica toate acele variabile referite în textul lambda-expresiei dar definite în afara ei.
- Contextul unei închideri functionale ramâne cel din momentul crearii închiderii functionale, cu exceptia variabilelor definite cu define, care ar putea fi înlocuite în timp.

Închideri funcţionale

- Când o închidere functionala este aplicata pe argumente, contextul salvat este folosit pentru a da semnificatie variabilelor libere din textul lambda-expresiei. Este vorba de contextul din momentul aplicarii, nu din momentul crearii închiderii.
- Un aspect remarcabil al inchiderilor functionale este ca pot fi folosite pentru a întârzia evaluarea.
- Plecând de la ideea ca o închidere functionala este o pereche text-context, iar textul nu este evaluat înainte de aplicarea lambda-expresiei pe argumente, consecinta este ca putem “închide” orice calcul pe care vrem sa îl amânam pe mai târziu într-o expresie (lambda () calcul) si sa provocam evaluarea exact la momentul dorit, aplicând aceasta expresie (aici pe 0 argumente).

Aplicatii ale inchiderilor functionale

1. Functii care pot imparti o stare privata
   - similar cu obiectele in POO (urmatorul exemplu)
2. Întârzierea evaluarii
   - Programarea cu structuri de date infinite, stream-uri
3. ...

Operaţii pe un cont privat (1)

```
(define (make-account amount)
 (let ([account amount])
  (define (store amount)
    (set! account (+ account amount)))
  (define (show) account)
  (define (withdraw amount)
    (printf "Withdraw ~a" (min account amount))
    (set! account (- account (min account amount))))
  (lambda (op (amount 0))
    (if (eq? op 'show)
        (show)
        (if (eq? op 'store)
            (store amount)
            (if (eq? op 'withdraw)
                (withdraw amount)
                (printf "unknown operation ~a" op)))))))
```

Operaţii pe un cont privat (2)

```
> ; crearea unui cont cu depozitul initial 1000
   (define acc1 (make-account 1000))
> acc1
#<procedure>

> (acc1 'store 300) ; depozit 300

> (acc1 'show)  ; interogare depozit pana acum
1300

> (acc1 'withdraw 400) ; retragere 400
Withdraw 400

> (acc1 'show)
900
```