Documentatie laborator 5 – Inchideri functionale. Evaluare lenesa. Fluxuri.

Inchideri functionale (closures)

Pentru a intelege intru totul notiunea de inchidere functionala, urmatoarele notiuni trebuie sa fie foarte clare:

Domeniu de vizibilitate a unei variabile (scope):

Variabilele sunt definite in program ca o pereche identificator-valoare prin mecanisme precum define, let, let*, letrec sau lambda. Semnificatia este ca valoarea poate fi referita prin intermediul identificatorului oriunde in program unde definitia este vizibila. Multimea punctelor din program unde definitia unei variabile este vizibila = domeniul de vizibilitate pentru variabila respectiva. Observatie: Presupunand ca am definit o variabila prin asocierea identificatorului x la valoarea 2, o definitie ulterioara precum (define x 3) nu schimba valoarea lui x, ci creeaza o noua variabila cu identificatorul x care se refera la valoarea 3. Cele 2 perechi (x 2) si (x 3) pot exista simultan in program, avand domenii diferite de vizibilitate.

Exemplu (cu 2 variabile referite prin identificatorul n, ale caror domenii de vizibilitate sunt marcate prin rosu, respectiv albastru):

```
(define n 2)

(define (fact n)
        (if     (< n 2)
                1
                (* n (fact (- n 1)))))
(fact n)
```

Legarea statica (lexicala) a variabilelor (lexical scoping):

Legarea este statica atunci cand domeniul de vizibilitate a variabilelor este controlat textual prin mecanisme specifice limbajului. De exemplu:

Pentru mecanismul let:

```
(define x 5)

(let ((x 1) (y x))          Variabila (x 1) definita cu let este vizibila doar in tot
        (display x)         corpul let-ului. In punctul definitiei (y x) ea nu este vizibila.
        (+ x y)))           Rezultat final: 6
```

Pentru mecanismul let*:

```
(define x 5)

(let* ((x 1) (y x))         Variabila (x 1) definita cu let* este vizibila doar in tot restul
        (display x)         textual al let*-ului. In punctul definitiei (y x) ea este vizibila.
        (+ x y)))           Rezultat final: 2
```

In cazul unui limbaj cu legare exclusiv statica, domeniul de vizibilitate a variabilelor este complet determinat dupa scrierea programului.

Legarea dinamica a variabilelor (dynamic scoping):

Legarea este dinamica atunci cand domeniul de vizibilitate a variabilelor este controlat dinamic, in functie de timp. Cu alte cuvinte, valoarea la care se refera un anumit identificator din textul unei expresii depinde de momentul in care alegem sa evaluam expresia. Pentru a intelege mai bine acest aspect, rulati urmatorul cod din fisierul lab5-doc.rkt:

```
(define a 2)
(define p (lambda () (+ a 5)))
(p)
(define a 6)
(p)
```

In Scheme legarea variabilelor este dinamica pentru variabilele definite cu define si statica in restul situatiilor (pentru variabilele definite cu lambda, let, let*, letrec…).

Context computational

Contextul computational al unui punct P din program la un moment t = multimea variabilelor vizibile in punctul P la momentul t.

Mai exact, contextul computational al unui punct P din program va contine perechi de tipul (identificator valoare). Valoarea asociata unui identificator poate varia in timp – exclusiv in cazul variabilelor definite cu define (intrucat acestea au legare dinamica).

Pentru exemplul de mai sus, in momentul definirii functiei p, punctul inrosit din program va avea in contextul sau perechea (a 2). Dupa apelul (define a 6), contextul aceluiasi punct din program este suprascris, perechea (a 2) fiind inlocuita cu perechea (a 6). Astfel rezultatul evaluarii (p) depinde de momentul evaluarii: prima evaluare (inainte de (define a 6)) va gasi in contextul expresiei evaluate legarea (a 2) si va intoarce rezultatul 7; a doua evaluare il va gasi pe a legat la valoarea 6 si va intoarce rezultatul 11.

Revenind la inchideri functionale

Conceptul de inchidere functionala a fost inventat in anii 60 si implementat pentru prima data in Scheme. Pentru a intelege acest concept, sa ne gandim se intampla in Scheme cand definim o functie, de exemplu functia de mai sus: (define p (lambda () (+ a 5))). Ceea ce face orice define este sa creeze o pereche identificator-valoare, asadar in acest caz se leaga identificatorul p la valoarea produsa de evaluarea λ-expresiei (lambda () (+ a 5)). Dar ce valoare produce evaluarea unei λ-expresii? Raspunsul este: Evaluarea oricarei λ-expresii produce o inchidere functionala.

O inchidere functionala este de fapt o pereche intre:

- textul λ-expresiei (aici (lambda () (+ a 5)))
- contextul computational in punctul de definitie a λ-expresiei (aici (a 2)).

Conform definitiei de mai sus, contextul unui punct P din program se refera la multimea variabilelor vizibile in punctul P. Ceea ce ne intereseaza de fapt sa salvam este multimea variabilelor libere in λ-expresia noastra, adica toate acele variabile referite in textul λ-expresiei dar definite in afara ei. Asa cum am explicat si mai sus, contextul unei inchideri functionale ramane cel din momentul crearii inchiderii functionale, cu exceptia variabilelor definite cu define, care ar putea fi inlocuite in timp.

Cand aceasta inchidere functionala este aplicata pe argumente (in exemplul nostru pe 0 argumente, dar tot inchidere functionala se numeste o λ-expresie cu oricate argumente) – contextul salvat in inchiderea functionala este folosit pentru a da semnificatie variabilelor libere din textul λ-expresiei. Am subliniat cuvantul cand pentru ca intotdeauna contextul folosit este contextul in momentul aplicatiei, nu cel din momentul crearii inchiderii functionale.

Un aspect remarcabil al inchiderilor functionale este ca ele pot fi folosite pentru a intarzia evaluarea. Plecand de la ideea ca o inchidere functionala este o pereche text-context, iar textul nu este catusi de putin evaluat inainte de aplicarea λ-expresiei pe argumente, consecinta este ca putem „inchide” orice calcul pe care vrem sa il amanam pe mai tarziu intr-o expresie (λ () calcul) si sa provocam evaluarea exact la momentul dorit aplicand aceasta expresie (aici pe 0 argumente).

Pentru a intelege mai bine cum functioneaza inchiderile functionale, experimentati cu exercitiile din lab5-doc.rkt.

Fluxuri implementate cu inchideri functionale

Pana acum am lucrat intensiv cu liste si cu functionale capabile sa manipuleze liste in cele mai uzuale situatii. Ce se intampla insa cand vrem sa lucram cu insiruiri oricat de lungi de elemente, de exemplu cu lista tuturor numerelor naturale? Nu putem memora toate numerele naturale intr-o lista pentru a le manipula ulterior. Sau ce se intampla daca vrem sa prelucram in n moduri o lista de foarte multe elemente? E oare cea mai buna solutie sa pasam de colo-colo o structura de date care ocupa atata memorie? Oare nu se va intampla de multe ori sa nu avem cu adevarat nevoie de toate elementele listei?

O solutie pentru aceasta problema sunt fluxurile – liste cu evaluare intarziata. Fluxurile exploateaza ideea inchiderilor functionale, definindu-se ca o pereche dintre primul element si, intr-o inchidere functionala, restul (inca necalculat al) fluxului. Inchiderea functionala actioneaza ca un motor care, cand este pornit (prin aplicatie de functie), se evalueaza la o noua pereche intre un prim element si, intr-o inchidere functionala, noul rest al fluxului. Astfel cerem dintr-un flux intotdeauna exact cate elemente sunt necesare.

O implementare si un exemplu gasiti in exercitiile din lab5-doc.rkt.

Evaluare lenesa

Evaluarea lenesa (lazy evaluation) este o tehnica de evaluare a expresiilor cu doua caracteristici esentiale:

- evaluarea unei expresii este amanata pana la momentul in care valoarea acelei expresii este necesara (call by need)
- evaluarea se realizeaza o singura data, rezultatul fiind salvat si refolosit de fiecare data cand mai este necesar (sharing)

Cu mici exceptii (precum functiile if, or, and), evaluarea in Scheme este aplicativa, insa limbajul permite implementarea evaluarii lenese cu ajutorul operatorilor delay si force. Cititi explicatiile si exemplele din lab5-doc.rkt pentru a intelege functionarea operatorilor delay si force, precum si pentru a studia felul in care sunt definiti operatorii pe fluxuri (cell, head, tail) intai cu inchideri functionale, apoi cu delay si force.

Observati ca:

- programele voastre merg indiferent de implementarea pentru cell, head si tail pe care o folositi. Aceasta se intampla pentru ca implementarea operatorilor este complet separata de utilizarea lor (ii putem folosi fara sa stim cum sunt implementati intern). Aceasta separare poarta numele de abstractizare si este o idee de baza in implementarea de sisteme complexe
- implementarea cu delay si force este superioara celei cu inchideri functionale intrucat expresiile intarziate cu delay vor fi evaluate maxim o data, pe cand cele intarziate cu inchideri functionale vor fi evaluate si reevaluate de fiecare data cand este nevoie