Ultima modificare: 25 martie

Tema 1 - Scheme Reflection

Introducere

Scopul acestei teme este implementarea reflexiei (eng. reflection) in Scheme. Reflexia este un mecanism pus la dispozitie de un limbaj de programare prin care un program scris in respectivul limbaj poate afla informatii la runtime despre entitatile sale componente si prin care isi poate modifica comportamentul.

Reflexia este comuna in special in randul limbajelor interpretate, precum python, ruby, php, javascript, Lisp, Scheme etc. dar si in randul celor compilate precum Java sau C#.

Asa cum s-a mentionat reflexia poate fi folosita atat pentru a obtine informatii despre obiectele din program la runtime cat si pentru a modifica comportamentul programului.

Spre exemplu, datele pe care le putem obtine in cazul limbajelor orientate obiect reprezinta informatii despre structura clasei unui obiect. Sa luam codul urmator:

```
import java.lang.reflect.Field;
import java.util.Collection;

public class SomeClass {

    public float x;
    public int y;
    public Collection<String> strCol;

    public static void main(String[] args) {
       SomeClass obj = new SomeClass();
       Class<? extends SomeClass> objClass = obj.getClass();
       System.out.println("Fields of " + objClass + " are: ");
       for (Field field : objClass.getFields()) {
             System.out.println(field.getName() + " with type " +
                  field.getType());
       }
    }
}
```

Aceasta va produce urmatorul output:

```
Fields of class SomeClass are:
x with type float
```

```
y with type int
strCol with type interface java.util.Collection
```

Intiatial se obtine clasa din care a fost instantiat un anumit obiect folosind metoda getClass. Aceasta tehnica poarta denumirea de introspectie de tip. Metoda getClass intoarce un obiect care incapsuleaza tipul obiectului in cauza. Folosind obiectul respectiv putem afla o multitudine de informatii despre clasa SomeClass. In exemplul dat am ales sa enumeram campurile accesibile ale clasei. Putem de asemenea sa obtinem metodele din clasa SomeClass si chiar sa apelam respectivele metode trimitand o lista de parametri.

Modificarea comportamentului se poate realiza in mai multe feluri. Un caz comun reprezinta transformarea unui string sau a unei structuri de date oarecare in cod executabil care poate fi ulterior rulat.

Iata un exemplu de cod javascript in care un string este interpretat.

```
var add1 = eval("function (x) { return x + 1; }");
var y = add1(1);
```

(NB: Javascript suporta functii ca valori de ordinul I, deci o scriere ok a codului de mai sus ar fi: var add1 = function(x) { return x + 1; };)

In cazul limbajelor din familia LISP (din care face parte si Scheme), reflexia este un mecanism care se poate realiza intr-un mod f. direct. Dupa cum probabil ati observat programele in Scheme sunt expresii organizate ca liste. Cum structura de date fundamentala este tot lista, e naturala existanta unei functii care sa ia o lista si sa evalueze codul continut in respectiva lista. Acea functie se numeste eval. Iata un exemplu in care scriem o functie care evalueaza o expresie cu variabile, data fiind un context reprezentat de perechi variabila - valoare:

```
(define (eval-in-context expr context)
  (eval (list 'let context expr)))
```

Un exemplu de aplicare a acestei funcții:

```
(eval-in-context '(- (+ a b) c) '((a 2) (b 3) (c 1))) => 4
```

Ce se întamplă mai exact:

- expresia (list ‘let context expr) de fapt creeaza expresia formata din liste imbricate:

```
(let ((a 2) (b 3) (c 1))
  (- (+ a b) c)))
```

- pe aceasta expresie se aplica eval, care evalueaza codul din structura data si se obtine rezultatul 4.

Reflexia poate fi utilizata in multe circumstante. Pe baza ei se pot implementa librarii si sisteme care necesita informatie de tip despre obiectele prelucrate la runtime. De asemenea, reflexia joaca un rol important in cadrul metaprogramarii.

Enuntul temei

In cadrul acestei teme vom implementa o forma de reflexie in Scheme cu scopul manipularii la runtime a functiilor.

Dupa cum stiti, crearea functiilor in scheme se face folosind lambda-expresii. Rezultatul unei astfel de constructii e reprezentat de un obiect de ordinul I complet opac pentru care singura operatie posibila este aplicarea pe un numar de argumente. Dorim sa extindem aceasta functionalitate si sa aflam mai multe informatii despre structura functiilor in timpul rularii programelor noastre.

Primul pas

In implementarea acestei cerinte veti pleca de la un API pe care il veti gasi in arhiva atasata temei. Vom descrie in continuare in ce consta acesta mini-librarie.

In pimul rand, pentru a putea face rationamente asupra functiilor va trebui sa avem access la λ-expresia de la care s-a pornit in crearea lor. Acest lucru implica faptul ca la crearea functiilor care suporta reflexie va trebui sa trimitem respectiva expresie “quotata” pentru a preveni evaluarea ei. De asemenea, dorim ca functiile cu reflexie sa se comporte ca si cum ar fi functii obisnuite - fara introducerea unui mod special de apelare, deosebit de cel al functiilor normale. Pentru a realiza aceste 2 constrangeri am ales urmatorul constructor:

```
(define (make-reflection-function name lambda-ex)                          ; 1
  (let ((real-function (eval lambda-ex)))                                  ; 2
        (lambda args                                                       ; 3
                (if     (and (not (null? args)) (null? (cdr args)))        ; 4
                        (cond                                              ; 5
                                ((equal? (car args) '__GET_NAME__) name)   ; 6
                                ((equal? (car args) '__GET_LAMBDA__) lambda-ex)  ; 7
                                (else (real-function (car args))))         ; 8
                (apply real-function args)))))                             ; 9
```

Constructorul primeste 2 parametri - un simbol care reprezinta numele functiei si lambda-expresia quotata asociata (pentru a preveni evaluarea). La linia a doua se face evaluarea lambda-expresiei legand astfel variabila real-function la obiectul functie obisnuit. Incepand cu linia a 3-a se defineste functia care ne intereseaza. Intrucat nu dorim sa ne limitam la functii care accepta doar un anumit numar de argumente functia in cauza va primi un numar variabil de argumente (Folosind constructia (lambda Nume expresie) definim functii cu numar

variabil de argumente). In primul rand testam daca functia a primit un singur argument. Daca da, testam daca s-a primit ca parametru simbolul special, ales de noi in mod conventional, __GET_NAME__ care va produce intoarcerea numelui functiei. Daca s-a primit __GET_LAMBDA__ vom intoarce λ-expresia corespunzatoare functiei. Altfel, aplicam functia reala pe unicul argument. Daca functia a primit 0, 2 sau mai multe argumente, folosim operatorul apply pentru a aplica functia reala pe argumentele trimise.

Spre exemplu, daca dorim sa cream functia add care aduna 2 numere scriem astfel:

(define add (make-reflection-function 'add '(lambda (x y) (+ x y))))
(add 2 3) ⇒ 5

Pentru a afla numele functiei apelam
(add '__GET_NAME__) ⇒ add

Si pentru aflarea lambda-expresiei apelam
(add '__GET_LAMBDA__) ⇒ (lambda (x y) (+ x y))

Dupa cum probabil ati observat acest mod de apelare este mai degraba nepotrivit pentru un limbaj elegant cum este Scheme. Pentru a repara acest lucru vom crea o serie de constructii de tip wrapper pe care le vom folosi ulterior.

Scheme ofera posibilitatea de a ne defini propria noastra sintaxa si ne vom folosi de aceasta facilitate pentru a simplica definirea functiilor cu reflexie.

```
(define-syntax define-r
  (syntax-rules ()
     ((_ name lambda-ex)
          (define name
               (make-reflection-function 'name 'lambda-ex)))))
```

Constructia de mai sus defineste cuvantul cheie “define-r” pe care il vom folosi pentru a defini functiile speciale. Pentru a intelege ce face define-syntax e suficient sa observati cum definim acum functia “add” de mai sus:

```
(define-r add
  (lambda (x y)
     (+ x y)))
```

Completam API-ul nostru simplu cu inca doua functii:

```
(define (get-name f) (f '__GET_NAME__))         ; Intoarce numele functiei

(define (get-lambda f) (f '__GET_LAMBDA__))     ; Intoarce lambda-expresia
functiei
```

Prin constructorul define-r si functiile get-name si get-lambda ascundem detaliile de implementare a librariei si expunem doar un tip de date abstract care se comporta ca o functie dar care in acelasi timp ofera informatii despre structura sa. In tema veti folosi DOAR define-r, get-name si get-lambda.

Cerinte

Folosind definitiile de mai sus va trebui sa implementati un numar de functii care opereaza pe tipul special definit mai sus. Pentru a nu creste complexitatea vom impune o serie de limitari asupra clasei de lambda-expresii asupra carora veti opera:

- Se considera ca exista doar urmatoarele functiile predefinite de mai jos. Aceasta inseamna ca in corpul unei functii se pot apela doar valorile de mai jos sau functia insasi, printr-un apel recursiv.
    - +, -, *, /, car, cdr, cons, null?, =, >, <, map si filter
- Aceste functii predefinite accepta un numar fix de argumente (nu variabil, asa cum permite Scheme in mod normal)
- Singurele expresii atomice permise sunt numerele intregi, valorile booleene #f si #t precum si simbolul special null care va semnifica lista vida.
- cons se considera a fi constructor de liste (nu de perechi).
- Singurele tipuri permise sunt numere, valori booleene si liste.
- Singurele constructii de limbaj permise sunt if, cond, lambda si let fara nume.

1) Detectia recursivitatii (3p)

Definiti functia recursion-type care ia ca parametru o functie si intoarce tipul de recursivitate pe care acea functie il manifesta. Acest tip poate fi unul dintre simbolurile:

- NON-RECURSIVE
- TAIL-RECURSIVE
- STACK-RECURSIVE
- TREE-RECURSIVE

De exemplu:
(recursion-type add) ⇒ NON-RECURSIVE

```
(define-r my-foldl
    (lambda (f acum L)
        (if     (null? L)
                acum
                (my-foldl f (f acum (car L)) (cdr L)))))
```

(recursion-type my-foldl) ⇒ TAIL-RECURSIVE

Functiile care sunt testate pentru aceasta parte nu vor contine map si filter.

2) Constant folding (2p)

Definiti o functie optimize care ia ca parametru o functie speciala si aplica optimizarea constant folding pe respectiva functie obtinand astfel o noua functie .

Constant folding inseamna evaluarea expresiilor care vor intoarce intotdeauna expresii constante. Spre exemplu, data functia

```
(define-r some-function
        (lambda (x)
                (if     (> x (+ 2 4))
                        (- 7 (car '(1 2 3)))
                        (cons x 4)))
(define f (optimize some-function))
(get-lambda f) ⇒
        (lambda (x)
                (if     (> x 6)
                        6
                        (cons x 4))))
```

Va trebui sa optimizati doar in urmatoarele cazuri:

- la un apel de functie cand toate argumentele sunt constante iar apelul nu este recursiv
- in cazul unei expresii cu **if** cand predicatul este o constanta #t sau #f si deci se poate decide pe ce ramura merge if-ul
- in cazul unei expresii **cond** cand se poate decide care conditie va fi cea care da rezultatul cond-ului.

Atat in cazul optimizarii pentru cond cat si pentru if inlocuiti expresia cu ramura care sigur se executa.

Pentru a va face viata mai usoara, folositi functiile **eval** si/sau **apply**.

Rezultatul intors de optimize trebuie sa fie o lambda-expresie.

3) Introspectie de tip (5p)

Definiti functia infer-type care ia ca parametru o functie si incearca sa determine signatura(tipul) functiei respective. Acest lucru inseamna ca va trebui sa determinati tipul fiecarui parametru din functie precum si tipul intors.

Dupa cum stiti, Scheme este un limbaj dinamic tipat, deci tipul unei variabile nu poate fi decis in cazul general decat la runtime, in functie de fluxul de executie. In cadrul acestei cerinte, puteti presupune ca Scheme este static tipat, si deci tipul fiecarei expresii poate fi decis la compilare. Acest lucru implica de asemenea ca listele sunt intotdeauna omogene (nu putem avea liste cu elemente de tipuri diferite).

Tipul fiecarei expresii poate fi:

TIP:

- NUMBER
- BOOLEAN
- Un simbol A - care reprezinta un tip variabil
- (LIST X) - unde X poate fi un tip oarecare
- Sau un tip de functie (FUN (TIP1 TIP2 … TIPN) TIP_RET)

Ca exemplu, tipul urmatoarelor expresii este:

- 3 ⇒ NUMBER
- (+ 1 2) ⇒ NUMBER
- (cons '(2 3) '((2 3 4) (1 2)) ⇒ (LIST (LIST NUMBER))
- (lambda (x y) (- x y)) ⇒ (FUN (NUMBER NUMBER) NUMBER)
- (lambda (x) x) ⇒ (FUN (A) A)
- (lambda (x y) (cons x (cons y null))) ⇒ (FUN (A A) (LIST A))

Tipul variabil se foloseste pentru functii pentru care tipul nu conteaza. Spre exemplu, fie functia identitate (lambda (x) x). Am putea spune ca tipul ei poate fi (FUN (NUMBER) NUMBER) sau (FUN (BOOLEAN) BOOLEAN) insa ea functieaza indiferent de tipul parametrului x. Din acest motiv folosim o variabila de tip ca sa-i exprimam signatura (FUN (A) A). In aceast caz am folosit A ca variabila (putem folosi orice nume atata vreme cat nu intram in conflict cu tipuri care chiar exista ca NUMBER, BOOLEAN, etc.). Specificam astfel ca functia identitate este generica.

Pentru o functie cu N argumente, infer-type va intoarce ceva de acest tip:
(FUN (TIP1 TIP2 … TIPN) TIP_RET)

Ca exemplu:

(infer-type add) ⇒ (FUN (NUMBER NUMBER) NUMBER)
(infer-type my-foldl) ⇒ (FUN ((FUN (A B) A) A (LIST B)) A)

Pentru a putea face inferenta pentru o functie data va trebui sa codificati in programul vostru signaturile pentru elementele predefinite:

+ ⇒ (FUN (NUMBER NUMBER) NUMBER)
- ⇒ (FUN (NUMBER NUMBER) NUMBER)
/ ⇒ (FUN (NUMBER NUMBER) NUMBER)
* ⇒ (FUN (NUMBER NUMBER) NUMBER)
cons ⇒ (FUN (A (LIST A)) (LIST A))
car ⇒ (FUN ((LIST A)) A)
cdr ⇒ (FUN ((LIST A)) (LIST A))
null? ⇒ (FUN ((LIST A)) BOOLEAN)
= ⇒ (FUN (NUMBER NUMBER) BOOLEAN)
> ⇒ (FUN (NUMBER NUMBER) BOOLEAN)
< ⇒ (FUN (NUMBER NUMBER) BOOLEAN)

map ⇒ (FUN ((FUN (A) B) (LIST A)) (LIST B))
#f ⇒ BOOLEAN
#t ⇒ BOOLEAN
orice numar ⇒ NUMBER
null ⇒ (LIST A)

Pentru deducere de tipuri exista o metoda clasica pe care o puteti folosi: algoritmul Hindley-Milner.

O introducere informala gasiti la urmatorul link: http://www.codecommit.com/blog/scala/what-is-hindley-milner-and-why-is-it-cool

Predare tema

O implementare se va considera adecvata daca indeplineste urmatoarele:

- cod scris modularizat
- implementare cat mai generica, abstracta
- cod scris in spiritul programarii functionale
  - folosirea functionalelor clasice (map, filter, fold)
  - folosirea functiilor ca valori de ordinul I

Se va scadea cu siguranta in urmatoarele circumstante:

- cod greu de urmarit, nedocumentat
- constructii imperative (din familia set!)

Se va incarca o arhiva cu numele **Grupa_Nume_Prenume_Tema1.zip** ce va contine sursele temei (Tema1.scm, completat si Reflection.ss, din arhiva, plus alte eventuale surse)

Deadline: miercuri, 30 martie, ora 7:00 am.

Mult Succes !