JScm: Compilator Scheme pentru masina virtuala Java

Constantin Stanciu
dsc@sundy.cs.pub.ro

Zoran Constantinescu
zoran@bit-soft.com

Universitatea "Politehnica" Bucuresti
Facultatea Automatica si Calculatoare

Motto: "Frunzele căzând"
"Se aştern una peste alta;"
"Ploaia loveşte în ploaie."

1 Introducere

Subiectul lucrarii este studierea posibilitatilor de realizare a unui compilator al limbajului Scheme. Scheme este un limbaj de programare orientat functional, apropiat de limbajele de programare functionale "moderne". El este un limbaj inrudit cu Lisp, insa in comparatie cu acesta este mult mai simplu. Pe linga aceasta are meritul sustinerii prin functii predefinite a unor tehnici avansate de programare (intreruperea si continuarea proceselor de calcul, evaluarea lenesa a expresiilor - "lazy evaluation"). Scheme implementeaza mecanismul de gestiune automata a memoriei. Programatorul nu se mai ocupa de alocarea si eliberarea de memorie pentru obiectele create in timpul executiei programului. Alocarea trebuie facuta explicit, dar este treaba mediului de executie unde se va aloca obiectul, programatorul neavind nici un control asupra acestei decizii. Eliberarea memoriei nu mai trebuie facuta de loc, o procedura de colectare a memoriei disponibile (garbage collector) va colecta toate obiectele ce nu mai pot fi referite prin program. Codul rezulat in urma compilarii a fost ales a fi byte-code pentru masina virtuala Java (Java Virtual Machine - JVM). Motivul alegerii este portabi- litatea oferita de Java la nivel de fisier cu continut executabil., precum si faptul ca si limbajul Java (resp. JVM) implementeaza mecanismul de garbage-collection. Astfel rularea unui program scris pentru aceasta masina virtuala este posibila pe orice sistem de operare pe care poate rula un emulator al unei astfel de masini virtuale.

2 Scheme extins

Limbajul Scheme implementat va fi un subset extins al standardului Scheme specificat in R4RS (Revised4 Report on the Algorithmic Language Scheme). Extensiile aduse limbajului Scheme sint:

* posibilitatea folosirii facilitatilor limbajului Java in limbajul Scheme, prin utilizarea claselor Java existente, in cadrul unui program Scheme;

(import "java.lang.io") (....)

* implementarea paradigmei Object Oriented in limbajul Scheme, permitind posibilitatea crearii unor obiecte in Scheme;

* folosirea in Scheme mecanismului de exceptii oferit de Java;

* folosirea in Scheme a mecanismului de threading oferit de Java.

Limbajul Scheme va implementa continuarea proceselor (call/cc), precum si eliminarea recursivitatii finale (tail-call recursivity). Implementarea celor doua mecanisme nu este posibila folosind o simpla translatare a expresiilor Scheme in bytecode Java.

3 Proiectarea compilatorului

Compilatorul va realiza translatarea programului sursa Scheme in program executabil Java prin intermediul mai multor pasi:

- Transformarea limbajului Scheme intr-un limbaj Scheme simplificat, in care sintaxa contine structurile de baza ale Scheme (lambda, define, set! ...). Toate constantele au tipurile specificate.
- Transformarea limbajului simplificat Scheme intr-un limbaj cu transmitere de continuari (Continuation Passing Style).
- Transformarea numelor variabilelor in referinte catre contextul local sau contextul global. Transformarea constantelor in referinte catre tabela de constante.
- Transformarea programului din limbajul CPS obtinut in cod "de asamblare" pentru masina virtuala Java.
- Transformarea codului rezultat in byte-code de Java, respectiv crearea de fisiere "class" Java, cu ajutorul unui asamblor de byte-code Java.

4 Transformarea Scheme $\longrightarrow$ Scheme simplificat

Aceasta transformare se face direct pe arborele sintactic al programului scheme initial. Transformarea are in vedere anumite tipuri de noduri din arbore si se realizeaza prin substitutie de subarbori.

Spre exemplu transformarea expresiilor de tip `cond` si `case`:

```
(cond (⟨test⟩ ⟨sequence⟩)
      ⟨clause2⟩ ...)
  ≡  (if ⟨test⟩
         (begin ⟨sequence⟩)
         (cond ⟨clause2⟩ ...))

(case ⟨key⟩
  ((d1 ...) ⟨sequence⟩)
  ...
  (else f1 f2 ...))
  ≡  (let ((key ⟨key⟩)
           (thunk1 (lambda () ⟨sequence⟩))
           ...
           (elsethunk (lambda () f1 f2 ...)))
       (cond ((⟨memv⟩ key '(d1 ...)) (thunk1))
             ...
             (else (elsethunk))))
```

5 Transformarea Scheme simplificat $\longrightarrow$ CPS

Conversia CPS îşi propune să aducă întreaga expresie scheme într-o forma "tail" (tail-form). În această formă funcţiile se transformă în proceduri care nu intorc rezultat ci transmit rezultatul pe care-l produc unei continuari. Fiecare functie se transforma intr-o continuare care preia un rezultat anterior calculeaza ceva pe baza lui si paseaza noul rezultat mai departe. Avantajul acestui tip de inlantuire a procedurilor este acela ca nu este necesar apelul unei proceduri cu punerea pe stiva a adresei de revenire ci se face un simplu salt la procedura apelata. Aceasta consecinta deriva din faptul ca nu avem nevoie sa ne intoarcem din vreo procedurar; cand aceasta se termina se face salt la noua procedura care este chiar contnuarea. Astfel un cod plin de apleluri recursive de functii, oricat de mare si incalcit ar fi, se transforma intr-un cod care nu produce modificari de stiva.

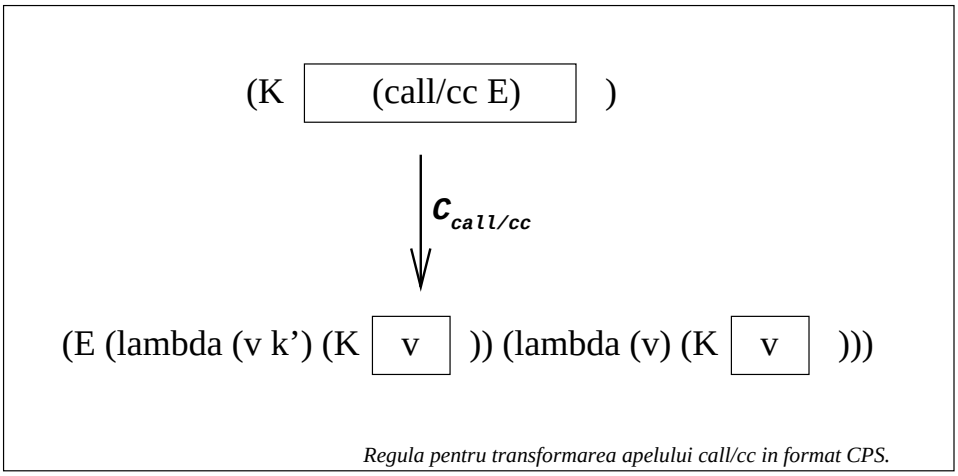
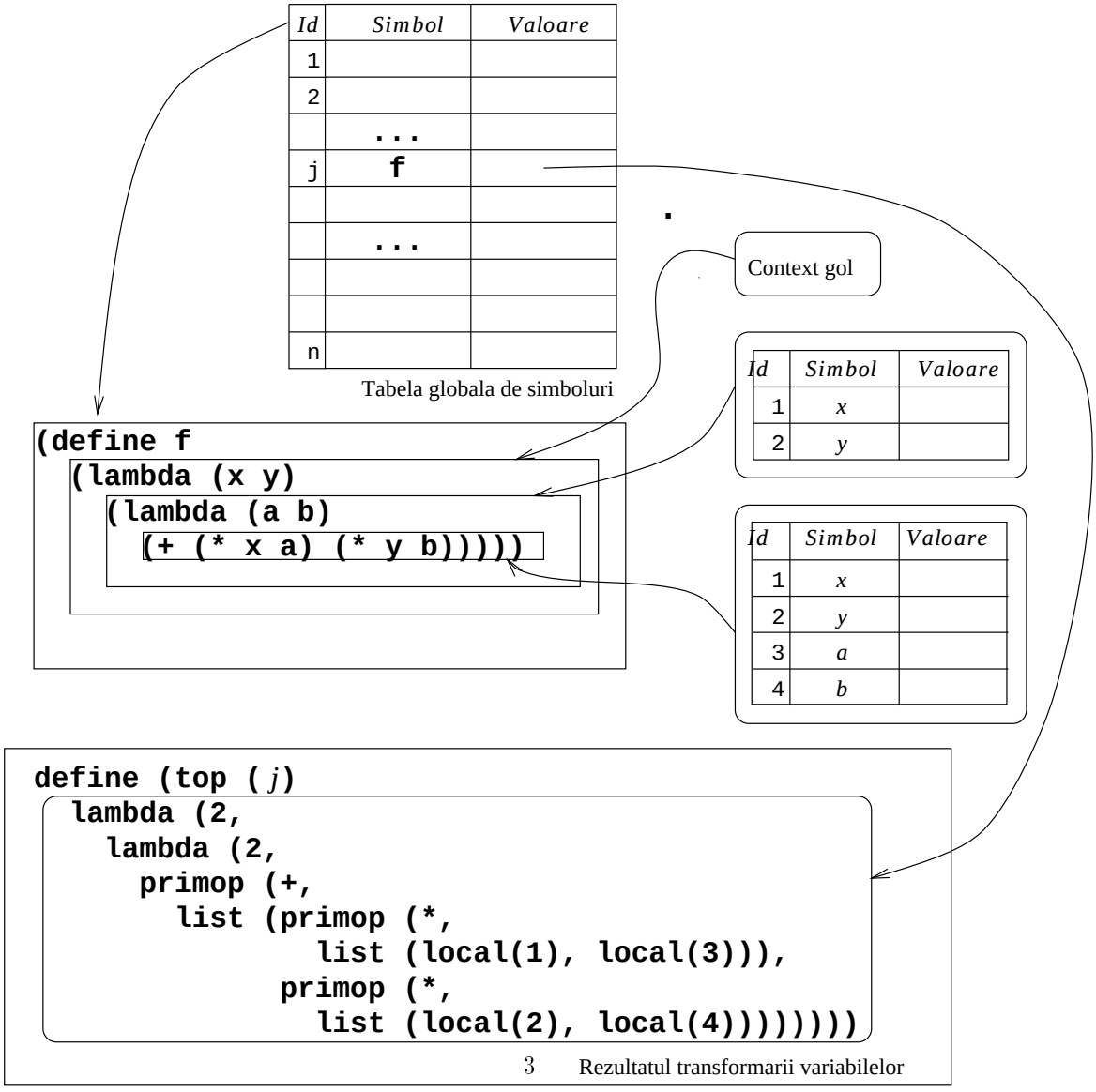


Figure 1: Transformarea CPS a formei call/cc.

5.1 Transforamrea numelor variabilelor



3 Rezultatul transformarii variabilelor

6 Generarea claselor Java pe baza programului CPS

Fiind un limbaj "dynamically typed", in Scheme verificarea tipurilor se face la executie si nu la compilare. Din acest motiv vom folosi pentru reprezentarea valorilor Scheme referinte la obiecte Java.

Fiecarui tip Scheme ii asociem cite o clasa Java, descendenta din clasa JScmObject. Fiecare din aceste clase va implementa doua tipuri de metode:

- corespunzatoare functiilor Scheme specifice tipului respectiv (ex. string-length pentru string; car pentru pair, list etc.);
- corepunzatoare functiilor Scheme aplicabile tuturor tipurilor (ex. char?, eq?, print - pentru afiasarea valorii unui tip etc.).

Tipurile Scheme si clasele corespunzatoare Java sint urmatoarele:

- boolean (#t, #f) $\Longrightarrow$ clasa JScmBoolean
- character $\Longrightarrow$ clasa JScmChar
- numbers $\Longrightarrow$ clasa JScmNumber (deocamdata doar intregi)
- pairs $\Longrightarrow$ clasa JScmPair
- list $\Longrightarrow$ clasa JScmList
- strings $\Longrightarrow$ clasa JScmString
- symbols $\Longrightarrow$ clasa JScmSymbol
- vectors $\Longrightarrow$ clasa JScmVector
- procedures $\Longrightarrow$ clasa JScmLambda
- unspecified $\Longrightarrow$ clasa JScmUnspec

Fiecare functie lambda definita de utilizator va primi un parametru suplimentar (pe linga parametrul suplimentar obtinut din transformarea CPS reprezentat de o continuare): un obiect JScmContext. Acesta reprezinta contextul de apel al functiei lambda. Noua functie lambda definita isi va crea si ea un context identic cu cel primit, dar in care va adauga variabilele locale (argumentele lui), pe care il va transmite in continuare expresiilor din corpul sau. Doar functiile lambda vor crea contexte noi, restul aplicarii de primop-uri vor transmite doar mai departe contextul primit. Un context va contine legarile de variabile locale functiilor lambda.

Toate apelurile de functii, in urma transformarii CPS, sint "tail form". Asta inseamna ca ele nu mai intorc controlul in functia apelanta (imediat superioara). Acest lucru permite implementarea functiilor recursive eliminind folosirea ineficienta a stivei. Cu alte cuvinte: apelul unei functii din cadrul altei functii nu se mai face cu apelul clasic de subrutina, ci cu un goto.

Astfel, urmatorul cod recursiv:

```
f:    stack=[arg1, arg2, ...]
  ...
  if <...>
    push arg1'
    push arg2'
    ...
    call f
  endif
  ...
  return   <- reface stiva, stergind argumentele primite
```

poate fi scris:

```
f:      stack=[arg1, arg2, ...]
  ...
  pop varg2
  pop varg1
  ...
  if <...>
    push arg1´
    push arg2´
    ...
    goto f
  endif
  ...
  return´    <- face doar intoarcerea din apelul functiei
                (echiv.  pop addr; goto addr
```

Problema in Java (ie. Java Virtual Machine) este urmatoarea: exista urmatoarele instructiuni pentru "apel" de functii, respectiv pentru transferul controlului:

1. `invokestatic`, `invokevirtual`, `invokenonvirtual`, `invokeinterface` pentru apelul metodelor unei clase Java;
2. `jsr`, `jsr_w` + `ret`, `ret_w` pentru apelul unor "minisubrutine" aflate in acelasi byte-code cu byte-code-ul metodei de unde se face apelul, ie. nu se pot face apeluri de subrutine din alte metode (object-oriented);
3. `goto`, `goto_w` pentru salt neconditionat in cadrul byte-code-ului aceleiasi metode.

In cazul primei metode avem doua optiuni:

1. putem considera fiecare functie Scheme CPS ca pe o clasa Lambda;
2. putem considera fiecare functie Scheme CPS ca pe o metoda a unei clase care cuprinde toate aceste functii.

In ambele situatii parametrii functiilor se transmit pe stiva, dar metoda va primi aceste valori ca variabile locale si nu pe stiva. Apelul uneia din instructiunile 'return, areturn' etc. la sfirsitul metodei va reface contextul dinaintea apelului metodei si pune pe stiva rezultatul apelului. Este clar ca din cadrul metodei nu se pot face modificari asupra stivei, deci aceste variante nu permit eliminarea recursivitatii.

O solutie pentru implementarea recursivitatii este folosirea combinata a instructiunilor 'jsr, ret, goto'. Acest lucru ne obliga ca fiecare functie Scheme (lambda definitie) sa reprezinte o "minisubrutina", apelabila prin jsr, dar si prin goto (exprimarea!). Toate aceste definitii de functii vor trebui implementate in cadrul `aceleiasi` metode. Apelul unei astfel de functii va fi posibila `doar` din cadrul aceleiasi metode, sau chiar din alta metoda (necesita o "miniinterfata" in cadrul metodei). Deci un top-level Scheme va fi implementat ca o clasa cu o metoda in care se definesc toate functiile.

Pe linga functiile pe care le defineste utilizatorul avem si functiile predefinite Scheme ('R4RS essential procedure'). Acestea sint considerate 'primop'-uri in transformarea CPS, ceea ce inseamna ca apelul lor nu va folosi nici o functie definita de utilizator. Aceste functii predefinite sint implementate intr-o clasa separata Java numita JScmFunc.

Implementarea unui program Scheme se face folosind o clasa Java cu o singura metoda. Aceasta va contine (printre altele) secvente de cod etichetate, corespunzatoare fiecarei expresii lambda. Apelul, respectiv saltul, unei astfel de expresii lambda se face folosind o instructiune byte-code 'lookupswitch', saltul la urmatoarea instructiune facindu-se pe baza unei key de selectie, cheie aflata pe stiva:

```
  switch_jsr:
    astore_x
    getstatic TopLevel.local_func
    getfield Lambda.lookup_key
    new Lambda
    dup
    invokenonstatic Lambda.<init> ()
    swap
  switch_goto:
    lookupswitch {
      k0:  lambda_0
      k1:  lambda_1
      k2:  lambda_2
      \ldots
    }
```

unde Lambda va fi o clasa Java care va contine cheia de salt al functiei, iar dupa caz si variabilele libere ale functiei:

```
  class Lambda extend java.lang.Object {
    public int lookup_key;
    Object free_vars[];

    public Lambda () {
      lookup_key = 0;
    }
    public void PutFreeVar (int i, Object x) {
      free_vars[i] = x;
    }
    public Object GetFreeVar (int i) {
      return free_vars[i];
    }
  }
```

Prima eticheta `switch_jsr` va fi folosita pentru apelul unei lambda expresii din cadrul unor expresii top-level, iar cea de-a doua `switch_goto` pentru apelul unei astfel de expresii din cadrul unei expresii in forma tail-form.

7 Asamblarea in byte-code

Asamblorul va realiza transformarea unui fisier sursa Java-bytecode intr-un fisier '.class', continind byte-code-ul corespunzator unei clase Java. Un exemplu de astfel de sursa 'Java asm' este urmatoarea secventa de cod:

```
class Hello2
extends java.lang.Object
{
  Method public static void main (java.lang.String[])
  max_stack 2
  max_locals 2
  {
    getstatic java.io.PrintStream java.lang.System.out
    ldc "Hello World!"
    invokevirtual void java.io.PrintStream.println(java.lang.String)
```

```
    return
  }

  Method void <init> ()
  max_stack 2
  max_locals 1
  {
    aload_0      /* "passed" in as variable 0 */
    invokenonvirtual void java.lang.Object.<init>()
    return
  }
}
```

References

[1] Friedman, Daniel P.; Wand, Mitchell; Haynes, Christopher T. *Essentials of Programming Languages*. MIT Press, 1992.

[2] Appel, Andrew W. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] William Clinger and Jonathan Rees (Editors). *Revised*^{4} *Report on the Algorithmic Language Scheme*. Available by anonymous ftp from `altdorf.ai.mit.edu`. 1991.

[4] Dybvig. *Three Implementation Models for Scheme*. University of North Carolina Computer Science Technical Report 87-011 [Ph.D. Dissertation], April 1987.