

## Program Optimization Using Abstract State Machines

Gabriel SOFONEA, Marian CRISTESCU, Laurențiu CIOVICĂ  
Catedra de Informatică Economică, Universitatea „Lucian Blaga” Sibiu

*Usually the result code of source code by a compiler is not necessary the best one, and can be improved to run faster or to use less memory. This kind of improvement is done in compiling phase after parsing. Some good techniques in optimization are in folding the constants, elimination of dead code, or improvement of the loops. Here it is considered the runtime overhead and present how can this be improved. The source is specific for object-oriented languages with late binding, where a name of method to be called is bound to method dynamically. It increases the computation time by a cost of traversing the class hierarchy each time a method is called.*

**Keywords:** optimization, abstract state machine (evolving algebra), class annotation.

### 1 Noțiuni preliminare

De obicei, codul rezultat produs din codul sursă de un compilator nu este întotdeauna cel mai bun; poate fi deseori îmbunătățit pentru se executa mai rapid sau pentru a folosi mai puțină memorie. Astfel de îmbunătățire este făcută în timpul fazei de compilare după parsare (derivare). Tehnicile bine cunoscute de optimizarea a mașinilor-independente includ strângerea constantelor, eliminarea codului inactiv (dead-code), eliminarea subexpresiilor comune și optimizarea diferitelor bucle. Detalii despre aceste metode pot fi găsite în [AHSEUL86].

În această lucrare se consideră depășirea timpului de execuție a unei surse și se va arăta modul cum se elimină aceasta. Sursa este specifică limbajelor orientate pe obiect cu legare întârziată, unde numele unei metode care va fi apelată este legată de metodă dinamic. Astfel crește timpul de calcul prin costul unei parcurgeri a ierarhiei clasei de fiecare dată când metoda este apelată.

În 3 vom transform algebra într-o formă care poate fi utilizată printr-o versiune modificată pentru ca programul să ruleze mai rapid prin legarea statică a unui nume de metodă cu metoda.

### 2. Colectarea informațiilor

Metoda de optimizare a programelor care urmează să fie discutate au nevoie de o colecție de clase receptor, care sunt folosite ca o intrare adițională pentru optimizare. Vom numi o colecție de clase receptor o

clasa de adnotare (însemnare). În general, problema de a determina clasa unei expresii date pentru un program dat și o intrare sunt nedecidabile.

Diferite metode de aflarea unei aproximații unei clase de adnotare au fost propuse anterior. Graver și Johnson au descris un sistem de scriere. Soluția lor necesită descrierea manuală a declarațiilor unor metode, și apoi tipurile de expresii pot fi reconstruite.

O complet diferită abordare a fost sugerată de Agesen și Hölze în [AGHO95]. Metoda lor, numită *feedback*, extrage clasele posibile ale receptorilor unui program  $P$  dintr-o rulare anterioară a lui  $P$ . În [AGHO95] este discutată și o comparație între tipul de reconstrucție și *feedback*.

În această secțiune vom defini algebra  $C_0$  obținută din algebra  $B_0$  prin adăugarea unui nou univers  $CLASS\_LIST$ , și trei funcții: *TypeOf*, *First* și *Method*. Elementele universului  $CLASS\_LIST$  reprezintă clasa de adnotări ca o listă de elemente ale universului  $CLASSES$ . Primul element a unei astfel de liste este dat de funcția:

$First : CLASS\_LIST \rightarrow CLASSES$

Se presupune că pentru fiecare element al universului  $EXPRESSIONS$  este dată o clasă de adnotare, ca și rezultat al unei metode menționate anterior. Pentru fiecare  $e \in EXPRESSIONS$  clasa de adnotare a lui  $e$  este dată de valoare a funcției

$TypesOf : EXPRESSIONS \rightarrow CLASS\_LISTS$

Dacă pentru  $e \in EXPRESSIONS$  reprezentând expresia mesajului unei clase a cărei receptor este determinat unic, ex. lista  $TypeOf(Receiver(e))$  are doar un element, atunci este posibil de determinat, ce metodă trebuie să fie invocată dacă  $e$  este evaluat. Pentru un astfel de  $e$  o funcție

$Method : EXPRESSIONS \rightarrow METHODS$

este definită. În secțiunea 2.1 vom da o definiție a valorii lui  $Method$ .

În plus, definim regula de tranziție  $R_2$  pentru  $C_0$ , care este o versiune modificată a regulii de tranziție  $R_1$ . Semantica definită de  $\langle B_0, R_1 \rangle$  este echivalentă cu semantica definită de  $\langle C_0, R_2 \rangle$  în sensul că dacă pentru unele programe  $P$ ,  $\langle B_0, R_1 \rangle$  definește semantica lui  $P$ , atunci  $\langle C_0, R_2 \rangle$  definește, de asemenea, o semantică echivalentă a lui  $P$ .

### 3. Transmiterea statică

#### 3.1. Transformarea metodelor

În această secțiune vom modifica algebra  $B_0$  pentru a putea folosi informațiile stocate de funcția  $TypeOf$ .

Presupunem că pentru un element  $e$  din universul  $EXPRESSIONS$  lungimea listei  $TypeOf(e)$  este 1, ex: clasa expresiei  $e$  este determinată unic. Presupunem, de asemenea, că  $e$  reprezintă un receptor al expresiei metodei, ex: există  $e' \in EXPRESSIONS$  astfel

încât  $Receiver(e') = e$ .

Atunci este ușor de a determina care metodă va fi apelată dacă prin calcul se ajunge la  $e'$ : este o valoare a termenului  $Method(Name(e'), First(TypeOf(e)))$  sau, dacă nu este definită cu numele  $Name(e')$  în clasa  $TypeOf(e)$ , atunci este căutată o metodă potrivită în clasa  $Superclass(TypeOf(e))$ , apoi în  $Superclass(Superclass(TypeOf(e)))$ , etc. Dacă o metodă este întâlnită în clasa  $C$ , atunci vom atribui valoarea termenului  $Method(Name(e'), C)$  termenului  $Method(e')$ . Altfel, dacă nici o metodă nu este găsită, atunci programul nu este corect.

O astfel de transformare executată pentru fiecare element din universul  $EXPRESSIONS$  face ca programele transformate să ruleze mai repede.

#### 3.2. Îmbunătățiri ale programului

Putem defini acum o regulă  $R_2$  care folosește funcțiile definite în algebra  $C_0$ . Definiția lui  $R_2$  este obținută din definiția lui  $R_1$  prin anumite modificări.

Pentru început se definește o nouă regulă  $MODIFIED\_LOOKUP$ . În  $R_2$  regula  $EXPRESSION\_EVALUATION$  diferă de regula  $EXPRESSION\_EVALUATION$  din  $R_1$ , atâta timp cât  $MODIFIED\_LOOKUP$  este folosit în loc de  $METHOD\_LOOKUP$ .

```

Rule MODIFIED_LOOKUP
if Method(CurrExp) ≠ undef then
    Method := Method(CurrExp)
    Receiver := Expression_value(Receiver(CurrExp), Current_state)
else
    METHOD_LOOKUP
endif
EndRule

Rule EXPRESSION_EVALUATION
if is not method then
    PRIMARY_EVALUATION
elseif Expression_value (Receiver ( CurrExp), Current_state) = undef then
    CurrExp:= Receiver ( CurrExp)
    elseif First_argument ( CurrExp) = undef then
        MODIFIED_LOOKUP
    elseif Expression_value (First_argument ( CurrExp), Current_state) = undef
then
    CurrExp:= First_argument ( CurrExp)
    else
        MODIFIED_LOOKUP
endif
EndRule

```

### 3.3 Echivalența semanticii

În această secțiune vom studia echivalența continuării semnaticilor date anterior, cât și versiunea optimizată cu trimitere (execuție) statică definită în secțiunea 3.1. Vom demonstra că rezultatele date de ambele semnatici sunt echivalente.

Fie  $P$  un program, și fie  $B_0$  o algebră definită pentru  $P$ . Fie  $C_0$  algebra definită de  $P$ .  $R_1$  și  $R_2$  sunt reguli de tranziție. Perechile  $\langle B_0, R_1 \rangle$  și  $\langle C_0, R_2 \rangle$  definesc două semantici operaționale ale lui  $P$ .

Apelăm la definiția echivalenței.

Dându-se un program  $P$ , două elemente  $b$  și  $c$  din  $B_0$  și, respectiv,  $C_0$  sunt echivalente  $b \equiv c$  dacă și numai dacă  $b$  și  $c$  reprezintă același nume, aceeași listă de nume, același obiect, aceeași variabilă, aceeași metodă, aceeași expresie sau aceeași stare din  $P$ .

Fie  $\Sigma$  o semnătură inclusă în semnăturile  $\Sigma_B$  și  $\Sigma_C$  a lui  $B_0$  și, respectiv,  $C_0$ . Trebuie să scriem că  $B_0$  și  $C_0$  sunt echivalente. Se observă că fiecare mulțime de reguli aplicabile  $R_1$  și  $R_2$  constau în cel mult o regulă de **extindere**, astfel că definiția echivalenței dintre elemente poate fi extinsă la elementele noi create (vezi definiția 3.2).

**Definiția 3.2** Fie  $C$  și  $D$  algebre astfel încât:

$$C \equiv_{\Sigma} D$$

Fie  $R$  un set de reguli actualizate asupra  $\Sigma$  incluzând și o regula *extend*. Fie  $c$  și  $d$  elemente create prin aplicarea  $R$  în  $C$  și respectiv  $D$ . Astfel avem  $c \equiv d$

Apoi rezultatele din aplicarea  $R$  în  $C$  și

- $B_j \models Receiver(Current\_expression(Current\_state)) \neq undef$
- $C_j \models Receiver(Current\_expression(Current\_state)) \neq undef$
- $C_j \models Method(Current\_expression(Current\_state)) \neq undef$

În astfel de caz valorile termenului  $Current\_expression(Current\_state)$  din  $B_i$  și  $C_i$  corespund unei expresii mesaj  $exp$  din programul  $P$ . Mai mult, metoda apelată de  $exp$ , poate fi determinată unic, iar elementul  $m$  corespunzător acestei metode este valoarea

*D sunt echivalente pe  $\Sigma$ .*

Definiția lui  $C_0$  este derivată din definiția lui  $B_0$ , prin adăugarea unei funcții adiționale *Method*. Deci, dacă  $\Sigma_B$  indică semnătura lui  $B_0$ , atunci

$$B_0 \equiv_{\Sigma_B} C_0$$

Fie  $R_1^*(B_0)$  calculul

$$R_1^*(B_0) = B_0, B_1, B_2, \dots, B_m, \dots$$

determinat de perechea  $\langle B_0, R_1 \rangle$ . Fie

$R_2^*(C_0)$  calculul

$$R_2^*(C_0) = C_0, C_1, C_2, \dots, C_n, \dots$$

determinat de perechea  $\langle C_0, R_2 \rangle$ .

Atâta timp cât  $R_1$  este diferit de  $R_2$ , evaluările  $R_1^*(B_0)$  și  $R_2^*(C_0)$  pot fi diferite. Mai departe considerăm aceste diferențe dintre  $R_1^*(B_0)$  și  $R_2^*(C_0)$ .

Presupunem ca  $B_i$  și  $C_j$  sunt astfel

$$B_i \equiv_{\Sigma_B} C_j$$

dar

$$R_1(B_i) \equiv_{\Sigma_B} R_2(C_j)$$

nu se păstrează. Aceasta implică, că au fost aplicate două mulțimi diferite de reguli, ex. Mulțimile de reguli aplicabile  $E_{B_i}(R_1)$  și  $E_{C_j}(R_2)$  sunt diferite. Notăm că există o singură diferență între  $R_1$  și  $R_2$ : în  $R_2$  LOOKUP\_METHOD este înlocuită de MODIFIED\_LOOKUP. Astfel, dacă

$$B_i \equiv_{\Sigma_B} C_j$$

și METHOD\_LOOKUP este aplicabilă lui  $B_i$  și MODIFIED\_LOOKUP este aplicabilă lui  $C_j$ , atunci

termenului

$Method(Current\_expression(Current\_state))$  din  $C_j$ .

Regulile METHOD\_LOOKUP și MODIFIED\_LOOKUP calculează o valoare a metodei constante *Method*, care corespund

metodei ce va fi apelată. METHOD\_LOOKUP (din  $R_1$ ) calculează *Method* în diferiți pași prin traversarea ierarhiei clasei. Acești pași formează o subșir a lui  $R_1^*(B_0)$

$$B_1, B_{i+1}, \dots, B_{i_k}$$

Pentru fiecare  $n, i \leq n \leq i_k$  avem

$$B_n \mid = \text{Method} = \text{undef}$$

și

$$B_{i_k} \mid = \text{Method} \neq \text{undef}$$

Observăm că METHOD\_LOOKUP aplicată lui  $B_n, i \leq n \leq i_k$  schimbă doar valoarea constantei *Class*. Aceasta implică, pentru fiecare  $n$  astfel încât  $i \leq n \leq i_k$

$$B_n \equiv_{\Theta} C_j$$

unde  $\Theta = \Sigma_B / \{\text{Class}\}$ .

În pasul final  $B_{i_k}$ , o valoare este atribuită lui *Method* prin METHOD\_LOOKUP. În  $R_2(C_0)$ , o valoare a lui *Method* este determinată în fiecare pas, care este deja calculată în  $R_2(C_j)$  prin evaluarea termenului

$Method(\text{Current\_expression}(\text{Current\_state}))$

, și atribuind valoarea sa lui *Method*. Dacă funcția *Method* este determinată corect (5.2), atunci valorile constantei *Method* din  $B_{i_k}$  și

$R_2(C_j)$  ar trebui să fie echivalente, deci

$$B_{i_k} \equiv_{\Sigma_B} R_2(C_j)$$

Știm că  $B_0 \equiv_{\Sigma_B} (C_0)$  deci două calcule:

$R_1^*(B_0)$  și  $R_2^*(C_0)$ , începând de la două

algebre echivalente. Mai mult, din propoziția 6.0.3 dacă  $B_i \equiv_{\Sigma_B} C_j$  atunci dacă mulțimea de reguli aplicabile  $E_{B_i}(R_1)$  și  $E_{C_j}(R_2)$  sunt egale, atunci  $R_1(B_i) \equiv_{\Sigma_B} R_2(C_j)$ .

Observăm că dacă  $E_{B_i}(R_1)$  și  $E_{C_j}(R_2)$  sunt egale atunci toți termenii din  $E_{B_i}(R_1)$  și  $E_{C_j}(R_2)$  sunt construiți peste  $\Sigma_B$ . Dacă aceste mulțimi de reguli aplicabile nu sunt egale, atunci  $R_1(B_i)$  și  $R_2(C_j)$  nu sunt echivalente peste  $\Sigma_B$ . Oricum, în astfel de caz există  $k$  nenegativ astfel încât

$$R_1^{(k)}(B_i) \equiv_{\Sigma_B} R_2(C_j)$$

Deci, calculele  $R_1^*(B_0)$  și  $R_2^*(C_0)$  devin echivalente din nou din  $R_1^k(B_i)$  și  $R_2(C_j)$ .

**Teorema 1** Calculul  $R_1^*(B_0)$  este finit dacă și numai dacă calculul  $R_2^*(C_0)$  este finit.

**Teorema 2** Fie calculele  $R_1^*(B_0)$  și  $R_2^*(C_0)$  finite. Fie  $R_1^\infty(B_0)$  ultima algebră a lui  $R_1^*(B_0)$ , fie  $R_2^\infty(C_0)$  ultima algebră a lui  $R_2^*(C_0)$ . Atunci

$$R_1^\infty(B_0) \equiv_{\Sigma_B} R_2^\infty(C_0)$$

Teorema 2 implică ca valoarea finală a constantei *Last\_value*, care reprezintă valoarea programului  $P$ , are interpretări echivalente în  $R_1^\infty(B_0)$  și  $R_2^\infty(B_0)$

#### 4. Concluzii

În această lucrare am prezentat o metodă de optimizare a programelor prin aplicarea semanticilor operaționale a limbajelor de programare orientate pe obiecte. O varietate de metode de definire a semanticilor a fost deja aplicată limbajelor orientate pe obiecte, descriind diferite aspecte ale unor astfel de limbaje, dar abordarea prezentată în lucrarea de față este mai abstractă în sensul de a da mai multe descrieri generale utilizând doar stări abstracte. Altă diferență este aceea că pentru a reduce modelul său acesta consideră o mulțime mai mică a limbajului fără ierarhia claselor care conduce la mai puține caracteristici dobândite, dar îi permite să dezvolte demonstrația unui sistem stil Hoare pentru algebrele sale.

#### Bibliografie

- [AGHO95] Agesen O., Hölzle U., „Type Feedback vs. Concrete Type Analysis: A Comparison of Optimization Techniques for Object-Oriented Languages”, Technical Report TRCS 95-04, Computer Science Department, University of California, Santa Barbara, March 1995;
- [AHSU86] Aho A.V., Sethi R., and Ullman J.D., *Compilers—Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986;

- [BÖRG95] E. Börger. *Annotated Bibliography on Evolving Algebras*. In E. Börger, editor, *Specification and Validation Methods*, pages 37–51. Oxford University Press, 1995;
- [GURE91] Gurevich Y., *Evolving Algebras. A Tutorial Introduction*. *Bulletin of EATCS*, 43:264–284, 1991;
- [GURE95] Gurevich Y., *Evolving Algebras 1993: Lipari Guide*. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995;
- [GURE00] Gurevich Y., *Sequential Abstract State Machines Capture Sequential Algorithms*. *ACM Transactions on Computational Logic*, 2000;
- [MÜLL94] Müller B., *A Semantics for Hybrid Object-Oriented Prolog Systems*. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, Elsevier, Amsterdam, the Netherlands, 1994.