Program Optimization Using Abstract State Machines

GABRIEL SOFONEA PhD candidate,
Computers Science Department
"Lucian Blaga" University of Sibiu
Sibiu, Dr. Ion Ratiu Street, no. 5-7,
ROMANIA

MARIAN-POMPILIU CRISTESCU PhD
Economic Informatics Department
"Lucian Blaga" University of Sibiu
Sibiu, Dr. Ion Ratiu Street, no. 5-7,
ROMANIA

Abstract: 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 Algebras), Annotation Class

1. Introduction in optimization

Usually the result code made 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. Details about these methods can be found in [2]

In this paper it is considered one source of runtime overhead and how to eliminate it. The source is specific for high level programming languages (object-oriented) with late-binding, where a name of method to be called it is bound to the method dynamically.

In this way the computation time will increase for parsing the class hierarchy each time this method is called.

In 3 the algebra will be transformed in a form that can be used by a modified version of R to run the program faster by statically binding a name of method with method.

2. Collecting information

The optimization method that will be discussed needs an collection of receivers' classes, which are used as an additional input for optimization. Will call this receivers' class, a *annotation class*.

In general, the problems to determine the class of a given expression for a given program and input are undecidable.

Different methods for finding an approximation of an annotation class were proposed. Graver and Johnson designed a type system. Their solution needs manual type declaration of methods, and than type of expressions can be reconstructed.

Completely different approach was suggested by Agesen and Hölze in [1].
Their method, named *feedback*, extracts receivers' possible classes of a program from a previous execution of P. The optimization of *P* is divided into two phases. In the first phase *P* is run and if some message expression of *P* is computed, then the class of the receiver of this expression is stored in memory. The second phase optimizes *P* using information stored in the previous run of *P*. In [1] a comparison between type reconstruction and feedback is discussed.

In this section will be defined an algebra C_0 obtained from algebra B_0 by adding a new universe CLASS_LIST, and three functions *TypeOf*, *First* and *Method*.
Elements of universe CLASS_LIST represents the annotation class as a list of elements from universe CLASSES. The first element of such a list is given by the function:

$$First : CLASS\_LIST \to CLASSES$$

Suppose that for each element from universe EXPRESSIONS an annotation class is given, e.g. as a result of one of methods mentioned above.
For each $e \in EXPRESSIONS$ the annotation class of *e* is given by the function value

$$TypesOf : EXPRESSIONS \to CLASS\_LISTS$$

If, for $e \in EXPRESSIONS$ representing the message expression a class of its receiver is determined uniquely, i.e. the list $TypeOf(REceiver(e))$ has only one element, than it is possible to determine, which method should be invoked if *e* is evaluated. For such *e* a function:

$$Method : EXPRESSIONS \to METHODS$$

it is defined. In section 2.1 will give a definition of the value of *Method*.

In addition, will be defined the transition rule R_2 for C_0, that is a modified version of transition rule R_1. Semantics defined by $< B_0, R_1 >$ is equivalent with semantics defined by $< C_0, R_2 >$ in the sense that for some programs *P*, $< B_0, R_1 >$ defines the semantics for *P*, than $< C_0, R_2 >$ defines, also, an equivalent semantics of *P*.

3. Static dispatch

3.1 Methods transformation

In this section the algebra B_0 will be modified to be possible to use the information stored by *TypeOf* function.

Suppose that for one element *e* from universe EXPRESSIONS, the length of list *TypeOf(e)* is 1, i.e. the expression class *e* is uniquely determined. Suppose, also that *e* represents a receiver of method expression, i.e. exists $e' \in EXPRESSIONS$ such that

$$\mathrm{Re}ceiver(e') = e$$

Than it is easy to determine which method will be called if computation comes to e': it is a value of the term $Method(Name(e'), First(TypeOf(e)))$ or, if no method with the name $Name(e')$ is in class *TypesOf(e)*, than a suitable method is search in class $Superclass(TypeOf(e))$, than in $Superclass(Superclass(TypeOf(e)))$, etc. If a method is found in class C, than the value of term $Method(Name(e'), C)$ will be assigned to the term $Method(e')$. Otherwise, if no method is found, then the program is not correct.

Such a transformation performed for each element from universe EXPRESSIONS makes that transformed programs to run faster.

3.2 Program improvements

We can define now a rule R_2 that uses the functions defined in algebra C_0.

Definition of R_2 is obtained from definition of R_1 by doing some modifications.

It is defined a new rule MODIFIED_LOOKUP. In R_2 the rule EXPRESSION_EVALUATION differs from rule EXPRESSION_EVALUATION from R_1, since MODIFIED_LOOKUP is used instead of METHOD_LOOKUP.

```
Rule MODIFIED_LOOKUP
if $Method(CurrExp) \neq undef$ 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 Equivalence of semantics

In this section we study the equivalence of the continuation semantics previously given, and also, its optimized version with static dispatch from section 3.1. We will demonstrate that the results given by both semantics are equivalent.

Let *P* be e program, and let be an algebra B_0 defined for *P*, and let C_0 be the algebra defined for *P*. R_1 and R_2 are transition rules. The pairs $< B_0, R_1 >$ and $< C_0, R_2 >$ defines two operational semantics of *P*.

Recall definition of equivalence:

Let *P* be a program, and *b* and *c* two elements from B_0, respectively C_0 are equivalents $b \equiv c$ if and only if *b* and *c* represents the same name, either the same list of names, either the same object, either the same variable, either the same method, expression or state from *P*.

Let Σ be a signature included in signatures Σ_B and Σ_C of B_0 and C_0. We shall write that B_0 and C_0 are equivalent up to Σ. Notice that each set of applicable rules R_1 and R_2 consists of most one rule *extend*, such that the definition of equivalence between elements can be extended for the new created elements (see definition 3.2)

Definition 3.2 *Let C and D be algebras such that:*

$$C \equiv_\Sigma D$$

Let be R a set of update rules over Σ including one extend rule. Let c and d be elements created by applying R in C and D. We have

$$c \equiv d$$

then results of applying R to C and D are also equivalent up to Σ

Definition of C_0 is derived from definition of B_0, by adding an additional function *Method.* So, if Σ_B denotes the signature of B_0 then

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

Let $R_1 * (B_0)$ be the computation

$$R_1 * (B_0) = B_0, B_1, B_2, ..., B_m, ...$$

determined by the pair $< B_0, R_1 >$.

Let $R_2 * (C_0)$ be the computation

$$R_2 * (C_0) = C_0, C_1, C_2, ..., C_n, ...$$

determined by the pair $< C_0, R_2 >$.

Since R_1 differs from R_2, the evaluations $R_1 * (B_0)$ and $R_2 * (C_0)$ can be different.

Bellow we consider these differences between $R_1 * (B_0)$ and $R_2 * (C_0)$.

Suppose that B_i and C_j such that

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

but

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

does not hold. This implies that two different set of rules have been applied, i.e. set of applicable rules $E_{B_i}(R_1)$ and $E_{C_j}(R_2)$ are different. Notice that exists only one difference between R_1 and R_2, in R_2 LOOKUP_METHOD is replaced by MODIFIED_LOOKUP. So, if

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

and METHOD_LOOKUP is applied to B_i and MODIFIED_LOOKUP is applied to C_j, than

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

In such case the values of term $Current\_expression(\text{Current\_state})$ from B_i and C_i corresponds to the message expression *exp* from *P*. Moreover, the method being called by *exp*, can be uniquely determined, and the element *m* corresponding to this method is the value of term $Method(Current\_expression(\text{Current\_state}))$ from C_j.

Rules METHOD_LOOKUP and MODIFIED_LOOKUP compute a value of constant method *Method*, which corresponds to the method that will be called.

METHOD_LOOKUP (from R_1) computed *Method* in different steps by parsing the class hierarchy. These steps forms a subset of $R_1 * (B_0)$

$$B_1, B_{i+1}, ..., B_{i_k}$$

For each $n, i \le n \le i_k$ we have

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

and

$$B_{i_k} \models \text{Method} \neq \text{undef}$$

Notice that METHOD_LOOKUP applied to $B_n, i \le n \le i_k$ changes only the value of the constant *Class.* This implies, that for each *n* such that $i \le n \le i_k$

$$B_n \equiv_\Theta C_j$$

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

In the final step B_{i_k}, a value is assigned to *Method* by METHOD_LOOKUP. In $R_2(C_0)$, a value of *Method* is determined in one step, and is already computed in $R_2(C_j)$ by evaluating the term $Method(Current\_expression(\text{Current\_state}))$, and assigning its value to *Method.* If function *Method* is determined correctly, then the values of the constant *Method* from B_{i_k} and $R_2(C_j)$ should be equivalent, hence

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

We know that

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

so two computations: $R_1*(B_0)$ and $R_2*(C_0)$, begin from two equivalent algebras.
Moreover, if

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

than, if the set of applicable rules $E_{B_i}(R_1)$ and $E_{C_j}(R_2)$ are equals, than

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

Notice that if $E_{B_i}(R_1)$ and $E_{C_j}(R_2)$ are equals than all terms from $E_{B_i}(R_1)$ and $E_{C_j}(R_2)$ are built over Σ_B. If these sets of applicable rules are not equals, than $R_1(B_i)$ and $R_2(C_j)$ are not equivalent over Σ_B. However, in this case exists *k* nonnegative such that

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

So, the computations $R_1*(B_0)$ and $R_2*(C_0)$ become equivalent again from $R_1^k(B_i)$ and $R_2(C_j)$.

Theorem 1 *The computation* $R_1*(B_0)$ *is finite if and only if the computation* $R_2*(C_0)$ *is finite.*

Theorem 2 *Let computations* $R_1*(B_0)$ *and* $R_2*(C_0)$ *be finite. Let* $R_1^\infty(B_0)$ *be the last algebra of* $R_1*(B_0)$, *let* $R_2^\infty(C_0)$ *be the last algebra of* $R_2*(C_0)$. *Than*

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

Theorem 2 implies that the final value of the constant *Last_value,* that represents the value if program *P*, has equivalent interpretations in $R_1^\infty(B_0)$ and $R_2^\infty(B_0)$.

4. Conclusions

In this paper we have presented an optimization method of the programs by applying operational semantics of an object-oriented programming language.

The problem 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 it, it is solved by the transformed algebra in a form that can be used by a modified version of R to run the program faster by statically binding a name of method with method.

*References***:**

[1] 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;

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers– Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[3] Y. Gurevich. *Evolving Algebras. A Tutorial Introduction*. *Bulletin of EATCS*, 43:264–284, 1991.

[4] Y. Gurevich. *Evolving Algebras* 1993: Lipari Guide. In E. Börger, editor, *Specifi-cation and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[5] Y. Gurevich. *Sequential Abstract State Machines Capture Sequential Algorithms*. *ACM Transactions on Computational Logic*, 2000. to appear.

[6] B. Müller. *A Semantics for Hybrid Object–Oriented Prolog Systems*. In B. Pehrsonand I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, Elsevier, Amsterdam, the Netherlands, 1994.

[7] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger,
editor, *Specification and Validation Methods*, pages 131–164. Oxford University, Press, 1995.

[JHRM01]John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation – 2nd ed*, Addison Wesley 2001.

[8] Ed Seidewitz, Mike Stark, *Reliable Object-Oriented Software: Applying Analysis and Design,* Cambridge University Press 97

[9] Wolf Zimmermann, Bernhard Thalheim, *Absract State Machine 2004, Advances in Theory and Practice*, Germany, 2004