An Approach to Polyvariant Binding Time Analysis for a Stack-Based Language

Yuri A. Klimov*

Keldysh Institute of Applied Mathematics, Russian Academy of Sciences
RU-125047 Moscow, Russia, yuklimov@keldysh.ru

Abstract. Binding time analysis (BTA) is used in specialization by means of partial evaluation method. Usual BTA only annotates a source program. Polyvariant BTA transforms a source program to an annotated one. Polyvariant BTA is known technique for functional languages. In this paper polyvariant BTA for a model imperative stack-based language is presented. It is described by means of building annotated control-flow graph for a source program.

1 Introduction

Partial evaluation is well known program specialization method [9]. Given values of *static* (known) arguments of a program, partial evaluation constructs a residual program — a specialized version of the source program, which on application to values of remaining *dynamic* arguments produces the same result as the source program applied to values of all arguments.

Offline partial evaluation stages the specialization in two phases: *binding time analysis* (BT-analysis, BTA) and residual program generating. BTA starts with a source program and a *binding time values* (BT-values) of all arguments and produces an annotated program.

There are two kinds of BT-analysis: *monovariant* and *polyvariant*. A monovariant BTA annotates the source program, does not transform it, whereas a polyvariant BTA generates a new annotated program. Monovariant BTAs are simple and efficient to implement [1,2,9,15,16]. Polyvariant BTAs [3,5,7,17] are more complex, but performs better result in many situations, when the same methods or variables are used in different contexts.

This paper consists of two part. First syntax and operational semantics of model imperative stack-based languages (SIL) are described. Then the polyvariant BTA for this language is presented by means of building annotated control-flow graph for a source program.

* Supported by Russian Foundation for Basic Research project No. 06-01-00574-a and No. 08-07-00280-a, and Russian Federal Agency of Science and Innovation project No. 2007-4-1.4-18-02-064.

Grammar

$p \in \text{Program} ::= instr^*$

$instr \in \text{Instruction} ::= \texttt{Pop} \mid \texttt{Dup} \mid \texttt{Swap} \mid \texttt{Const}(c) \mid \texttt{Goto}(n) \mid \texttt{IfGoto}(n)$
$\texttt{Unary}(op) \mid \texttt{Binary}(op) \mid \texttt{LoadVar}(n) \mid \texttt{StoreVar}(n)$

Fig. 1. Abstract syntax of SIL-programs

2 Imperative Stack-Based Language

For describing the polyvariant BTA the imperative stack-based language (SIL) is used. SIL is a very simple stack language (fig. 1). A program at this language is just sequence of instructions (no methods and invoke instructions) with conditional and unconditional goto instructions (`IfGoto(m)` and `Goto(m)`). Other instructions are load (`LoadVar(n)`) and store (`StoreVar(n)`) data from stack to local variables, operations with data on stack (`Unary(op)` and `Binary(op)`) and simple stack operations (`Pop`, `Dup`, `Swap`).

Instructions

$$\texttt{Pop} \vdash_{int} (x : st, \sigma) \to (st, \sigma) \qquad \texttt{Dup} \vdash_{int} (x : st, \sigma) \to (x : x : st, \sigma)$$
$$\texttt{Swap} \vdash_{int} (x_1 : x_2 : st, \sigma) \to (x_2 : x_1 : st, \sigma) \qquad \texttt{Const}(c) \vdash_{int} (st, \sigma) \to (c : st, \sigma)$$
$$\texttt{Unary}(op) \vdash_{int} (x : st, \sigma) \to (op(x) : st, \sigma)$$
$$\texttt{Binary}(op) \vdash_{int} (x_1 : x_2 : st, \sigma) \to (op(x_1, x_2) : st, \sigma)$$
$$\texttt{LoadVar}(n) \vdash_{int} (st, \sigma) \to (\sigma(n) : st, \sigma)$$
$$\texttt{StoreVar}(n) \vdash_{int} (x : st, \sigma) \to (st, \sigma[n \mapsto x])$$

Instructions with control point

$$\texttt{Goto}(n) \vdash_{int} (m, (st, \sigma)) \to (n, (st, \sigma))$$
$$\texttt{IfGoto}(n) \vdash_{int} (m, (0 : st, \sigma)) \to (m + 1, (st, \sigma))$$
$$\texttt{IfGoto}(n) \vdash_{int} (m, (1 : st, \sigma)) \to (n, (st, \sigma))$$

$$\frac{instr \vdash_{int} (st, \sigma) \to (st', \sigma')}{instr \vdash_{int} (m, (st, \sigma)) \to (m + 1, (st', \sigma'))}$$

Program

$$\frac{p(m) \vdash_{int} (m, (st, \sigma)) \to (m', (st', \sigma'))}{p \vdash_{int} (m, (st, \sigma)) \to (m', (st', \sigma'))}$$

$$\frac{p \vdash_{int} (0, (st, \sigma_{init})) \to^* (\text{length}(p), (st', \sigma))}{p \vdash_{int} st \Rightarrow st'}$$

Fig. 2. Operational semantics of SIL-programs

The data in SIL is integer numbers. Nevertheless it is easy to extend data by other data (double numbers or boolean values). Operation *op* in `Unary(op)` and `Binary(op)` can be any operation with integer numbers, including, but not limited to, addition (+), subtraction (−), multiplication (×), compare (<, ≤, >, ≥), negate and etc.

The semantic of SIL is straightforward and it is described at fig. 2. A SIL-program is evaluated by steps. Each step changes a state. Each state $(m, (st, \sigma))$ has three parts: m — number of current instruction (control point), st — stack of values, σ — mapping from local variables to their values.

A computation of a program begins from initial state $(0, (st, \sigma_{init}))$, where st — program arguments, and σ_{init} — mapping from local variables to initial value 0. At each step state is changed in according to the rules. When number of current instruction becomes equal to length of the program then evaluation of this program is finished. Values at a stack are results of this program. If number of current instruction becomes more than length of the program or no rules can be applied then evaluation of this program is terminated with error.

The SIL is similar to stack-based languages described in [2] or [15]. It contains same instruction set except array instructions and method invoke instructions.

3 Binding Time Analysis

The goal of BTA is to divide all instructions in two classes: static (**S**) and dynamic (**D**). Static instructions will be evaluated during residual program generating, dynamic instructions will be put to residual program.

The presented method of building annotated program is close to the Supercompilation [18]. It uses driving and whistling for building possibly infinity binding time tree (BT-tree) and for reducing it to finite binding time graph (BT-graph) respectively.

3.1 Driving

BT-tree is a tree with annotated instructions at nodes and with binding time states (BT-states) at ridges (fig. 4). BT-tree is similar to control-flow graph without ridge to previous nodes, each ridge is going to a new node. This BT-tree can be applied to arguments values like usual program. The BT-tree is fully equivalent to the source program: on application to values it produces the same result as the source program applied to same values.

BT-state is a state with binding time values (BT-values) **S** and **D** instead of usual values. Bold style for binding time values and variables are used below: **S** and **D** are BT-values, **x** is a binding time variable (BT-variable) that rages over BT-values **S** and **D**.

Building of BTA tree begins with a initial BT-state $(0, (st, \sigma_{init}))$, where st — BT-values of arguments (**S** corresponds to known arguments and **D** corresponds to unknown during specialization arguments) and σ_{init} — mapping from local variables to initial BT-value **S**.

Lifting instruction

$$\texttt{Lifting}(n) \vdash_{int} (st, \sigma) \to (st, \sigma)$$

Instructions

$$\texttt{Pop} \vdash_{bta} (\mathbf{x} : st, \sigma) \to \langle \texttt{Pop}^{\mathbf{x}}; (st, \sigma)\rangle$$
$$\texttt{Dup} \vdash_{bta} (\mathbf{x} : st, \sigma) \to \langle \texttt{Dup}^{\mathbf{x}}; (\mathbf{x} : \mathbf{x} : st, \sigma)\rangle$$
$$\texttt{Swap} \vdash_{bta} (\mathbf{x} : \mathbf{x} : st, \sigma) \to \langle \texttt{Swap}^{\mathbf{x}}; (\mathbf{x} : \mathbf{x} : st, \sigma)\rangle$$
$$\texttt{Swap} \vdash_{bta} (\mathbf{S} : \mathbf{D} : st, \sigma) \to \langle \texttt{Swap}^{\mathbf{S}}; (\mathbf{D} : \mathbf{S} : st, \sigma)\rangle$$
$$\texttt{Swap} \vdash_{bta} (\mathbf{D} : \mathbf{S} : st, \sigma) \to \langle \texttt{Swap}^{\mathbf{S}}; (\mathbf{S} : \mathbf{D} : st, \sigma)\rangle$$
$$\texttt{Const}(c) \vdash_{bta} (st, \sigma) \to \langle \texttt{Const}(c)^{\mathbf{S}}; (\mathbf{S} : st, \sigma)\rangle$$
$$\texttt{Unary}(op) \vdash_{bta} (\mathbf{x} : st, \sigma) \to \langle \texttt{Unary}(op)^{\mathbf{x}}; (\mathbf{x} : st, \sigma)\rangle$$
$$\texttt{Binary}(op) \vdash_{bta} (\mathbf{x} : \mathbf{x} : st, \sigma) \to \langle \texttt{Binary}(op)^{\mathbf{x}}; (\mathbf{x} : st, \sigma)\rangle$$
$$\texttt{Binary}(op) \vdash_{bta} (\mathbf{S} : \mathbf{D} : st, \sigma) \to \langle \texttt{Lifting}(0)^{\mathbf{D}}, \texttt{Binary}(op)^{\mathbf{D}}; (\mathbf{D} : st, \sigma)\rangle$$
$$\texttt{Binary}(op) \vdash_{bta} (\mathbf{D} : \mathbf{S} : st, \sigma) \to \langle \texttt{Lifting}(1)^{\mathbf{D}}, \texttt{Binary}(op)^{\mathbf{D}}; (\mathbf{D} : st, \sigma)\rangle$$
$$\texttt{LoadVar}(n) \vdash_{bta} (st, \sigma) \to \langle \texttt{LoadVar}(n)^{\sigma(n)}; (\sigma(n) : st, \sigma)\rangle$$
$$\texttt{StoreVar}(n) \vdash_{bta} (\mathbf{x} : st, \sigma) \to \langle \texttt{StoreVar}(n)^{\mathbf{x}}; (st, \sigma[n \mapsto \mathbf{x}])\rangle$$

Instructions with control point

$$\texttt{Goto}(n) \vdash_{bta} (m, (st, \sigma)) \to \langle \texttt{Goto}(n)^{\mathbf{S}}; (n, (st, \sigma))\rangle$$

$$\texttt{IfGoto}(n) \vdash_{bta} (m, (\mathbf{x} : st, \sigma)) \to \langle \texttt{IfGoto}(n)^{\mathbf{x}}; (n, (st, \sigma)), (m+1, (st, \sigma))\rangle$$

$$\frac{instr \vdash_{bta} (st, \sigma) \to \langle instrs; (st', \sigma')\rangle}{instr \vdash_{bta} (m, (st, \sigma)) \to \langle instrs; (m+1, (st', \sigma'))\rangle}$$

Program

$$\frac{p(m) \vdash_{bta} (m, (st, \sigma)) \to \langle instrs; brs\rangle}{p \vdash_{bta} (m, (st, \sigma)) \to \langle instrs; brs\rangle}$$

Fig. 3. Trace semantics for binding time trees

For each BT-state one of rules (fig. 3) is applied. The rule shows an annotated instruction at new node and one or two new BT-states at ridges in subject to current BT-state and instruction. For example, if there are BT-state $(m, (\mathbf{x} : st, \sigma))$ and current instruction `IfGoto(`n`)`, then it is needed to add new node with two new ridges to BT-tree. The node must contain annotated instruction $\texttt{IfGoto}(n)^{\mathbf{x}}$ and the ridges must contain BT-states $(n, (st, \sigma))$ and $(m+1, (st, \sigma))$.

In come cases new instruction `Lifting(`n`)` is added to a BT-tree. This instruction tells residual program generator that static (known) value in stack at depth n must be residualized be means of generating `Const(`c`)` instruction and some stack instructions. For interpretation `Lifting(`n`)` instruction means no operation.

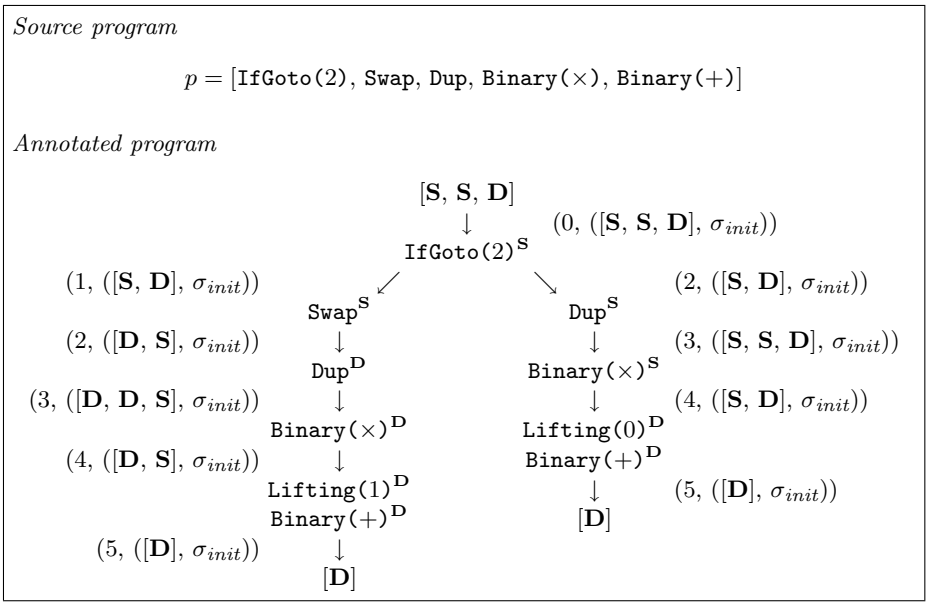

Fig. 4. BT-tree for the program $p(x, y, z) =$ if x then $y^2 + z$ else $z^2 + y$; x and y are static (known), z is dynamic (unknown).

3.2 Whistling

During BT-tree building all BT-states are checked for conjunction. If two BT-states (may be at different branches) are equal (whistling) when nodes at the end of this ridges are be merged into the new node. It is permitted because BT-tree constructed from some BT-state depends on this BT-state only.

Building of BT-graph is ending because there are only finite numbers of all possible BT-states for a SIL-program. Residual BT-graph is graph representation of annotated program. Residual Program Generating for such annotated program is identical to [2].

3.3 Example

Let's consider a small program p (fig. 4). The polyvariant BTA produces a BT-graph represented at fig. 4.

This BT-graph contains two instructions `Binary(×)` with different BT-annotations according to different BT-annotations of stack at the same program point. At the left hand size instruction `Binary(×)` annotated as dynamic (**D**), while at the right hand size instruction `Binary(×)` annotated as static (**S**).

This means that if first argument of p is false during specialization then instruction `Binary(×)` will be residualized. In other case instruction `Binary(×)` will be evaluated during residual program generating.

4 Related Work

In many prior works monovariant BTAs [9] for functional languages are described. In [3,7,17] polyvariant BTAs for functional languages are presented. In minority works [1,2,15,16] monovariant or polyvariant BTAs for imperative languages are considered [13].

L. O. Andersen [1] uses C language. P. Bertelsen [2] and H. Masuhara and A. Yonezawa [15] describe monovariant BTA for various subsets of stack-based Java Byte Code [8]. In both papers simple stack-based language like SIL (which is described in this paper) is uses: in [2] SIL is extended with array instruction and in [15] SIL is extended with method invocation instruction.

U. P. Schultz [16] introduces monovariant BTA some subset of Java language [8]: object-oriented but not stack-based language. Also he suggests some polyvariant (class polyvariant and method polyvariant) extensions of BTA. N. H. Christensen and R. Glück [5] present polyvariant BTA for flowchart imperative language.

Presented BTA extends prior monovariant BTAs for stack-based language by introducing control-point polyvariant, stack polyvariant and environment polyvariant annotation method. It uses infinite control-flow tree for building finite annotated graph. This new method bases on ideas of Supercompilation [18]. It is possible to enhance this method for a object-oriented stack-based language [10,11,12].

5 Conclusion

In this paper polyvariant BTA for simple stack-based language is introduced. This method is fully automatic and it is used in specializer CILPE [4,14].

In some cases polyvariant BTA can produce a huge residual program. I would like to investigate extensions of polyvariant BTA for producing a program of reasonable size. Another direction of research is to enhance this method for object-oriented stack-based languages such as Java Byte Code (Java platform) [8] and Common Intermediate Language (Microsoft .NET platform) [6] which are used in popular virtual machines.

References

1. L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, Computer Science Department, University of Copenhagen, 1994. DIKU Technical Report 94/19.
2. P. Bertelsen. Binding-time analysis for a JVM core language. Unpublished note; available from `http://www.dina.kvl.dk/$\sim$pmb`. 1999.
3. M. A. Bulyonkov. Extracting polyvariant binding time analysis from polyvariant specializer. *Partial evaluation and semantics-based program manipulation. Proceedings*, 59–65. ACM Press, 1993.

4. A. M. Chepovsky, An. V. Klimov, Ar. V. Klimov, Yu. A. Klimov, A. S. Mishchenko, S. A. Romanenko, S. Yu. Skorobogatov. Partial Evaluation for Common Intermediate Language. M. Broy and A. V. Zamulin (eds.), *Perspectives of Systems Informatics. Proceedings*, LNCS 2890, 171–177. Springer-Verlag, 2003.
5. N. H. Christensen, R. Glück Offline partial evaluation can be as accurate as online partial evaluation. *ACM Transactions on Programming Languages and Systems*, vol. 26(1), 191-2004. ACM Press, 2004.
6. Common Language Infrastructure. `http://msdn2.microsoft.com/en-us/netframework/aa569283.aspx`.
7. C. Consel. Polyvariant binding-time analysis for applicative languages. *Partial evaluation and semantics-based program manipulation. Proceedings*, 66–77. ACM Press, 1993.
8. Java Virtual Machine. `http://java.sun.com/docs/books/jvms/`.
9. N. D. Jones, C. K. Gomard, P. Sestoft. Partial Evaluation and Automatic Compiler Generation. C.A.R. Hoare Series, Prentice-Hall, 1993.
10. Yu. A. Klimov. Polyvariant binding time analysis in specializer CILPE for Common Intermediate Language of Microsoft .NET planform. *Microsoft technologies in theory and practice of programming. Proceedings*, 128. 2005. (In Russian)
11. Yu. A. Klimov. About polyvariant binding time analysis for specializer for object-oriented language. *Scientific service in the Internet: technology of distributed computations. Proceedings*, 89–91. 2005. (In Russian)
12. Yu. A. Klimov. Residual program generator and correctness of specializer for object-oriented language. *Scientific service in the Internet: technology of parallel programming. Proceedings*, 137-140. 2006. (In Russian)
13. Yu. A. Klimov. Program specialization for object-oriented languages by partial evaluation: approaches and problems. Preprint No. 12, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 2008. (In Russian)
14. Yu. A. Klimov. Specializer CILPE: examples of object-oriented program specialization. Preprint No. 30, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 2008. (In Russian)
15. H. Masuhara, A. Yonezawa. Run-time Program Specialization in Java Bytecode. *Workshop on Systems for Programming and Applications. Proceedings.* 1999.
16. U. P. Schultz. Object-Oriented Software Engineering Using Partial Evaluation. PhD thesis, University of Rennes I, Rennes, France, December 2000.
17. P. Thiemann, M. Sperber. Polyvariant expansion and compiler generators. D. Bjørner, M. Broy, I. V. Pottosin (eds.), *Perspectives of Systems Informatics. Proceedings*, LNCS 1181, 285–296. Springer-Verlag, 1996.
18. V. F. Turchin. The concept of a supercompiler. ACM Transactions on Programming Languages and Systems, 8(3):292–325. ACM Press, 1986.