A Java Supercompiler and Its Application to Verification of Cache-Coherence Protocols

Andrei V. Klimov*

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq., Moscow, 125047, Russia
klimov@keldysh.ru

Abstract. The Java Supercompiler (JScp) is a specializer of Java programs based on the Turchin's supercompilation method and extended to support imperative and object-oriented notions absent in functional languages. It has been successfully applied to verification of a number of parameterized models including cache-coherence protocols. Protocols are modeled in Java following the method by G. Delzanno and experiments by A. Lisitsa and A. Nemytykh on verification of protocol models by means of the Refal Supercompiler SCP4. The part of the supercompilation method relevant to the protocol verification is reviewed. It deals with an imperative subset of Java.

Keywords: specialization, verification, supercompilation, object-oriented languages, Java.

1 Introduction

Program specialization methods — partial evaluation [10], supercompilation [23,24,25], mixed computation [8], etc. — have been first developed for functional and simplified imperative languages. Later the time has come for specialization of more complex practical object-oriented languages.

There are already a number of works on partial evaluation of imperative and object-oriented languages [3,4,5,15,21]. However, to the best of our knowledge, our work is the first one on supercompilation of a practical object-oriented language [9,11,14]. Inspired by far-reaching results by Alexei Lisitsa and Andrei Nemytykh on verification of protocol models by means of the Refal Supercompiler SCP4 [16,17,18], we extended the Java Supercompiler with the elements of the supercompilation method that were needed to reproduce the results in Java [12] (namely, with restrictions on configuration variables of integral types).

Specialization of operations on objects in JScp is discussed in another paper [11]. Since objects are not used in the protocol models coded in Java, in this paper we review supercompilation of the imperative subset of Java.

* Supported by Russian Foundation for Basic Research projects Nos. 08-07-00280-a, 09-01-00834-a, and 09-07-13598-ofi_ts, and Russian Federal Agency of Science and Innovation project No. 2007-4-1.4-18-02-064.

A. Pnueli, I. Virbitskaite, and A. Voronkov (Eds.): PSI 2009, LNCS 5947, pp. 185–192, 2010.

A novelty of this part of the supercompilation method implemented in JScp is that *breadth-first* unfolding of the graph of configurations and recursive construction of the residual code of a statement from the residual codes of nested statements is used rather than *depth-first* traversal of configuration as in other known supercompilers. Another contribution of this paper is reproduction of the results of the experiment on verification of protocols by another supercompiler (JScp instead of SCP4) for a rather different language (the object-oriented Java instead of the functional Refal). This confirms the result is based on the essence of supercompilation rather than on technical implementation details. As a consequence of the experiment the part of supercompilation method relevant to the verification of protocols has been uncovered.

This paper is an extended abstract of the longer version published in the PSI'09 preproceedings [13]. It is organized as follows. In Section 2 the part of the Java supercompilation method that is relevant to verification of protocols is reviewed. In Section 3 experiments on verification of protocol models are described. In Section 4 we conclude.

2 Java Supercompilation

The notion of a configuration While an interpreter runs a program on a ground data, a supercompiler runs the program on a set of data.

A representation of a subject program state in a supercompiler is referred to as a *configuration*. We follow the general rule of construction of the notion of a configuration in a supercompiler from that of the program state in an interpreter that reads as follows: add *configuration variables* to the data domain, and allow the variables to occur anywhere where an ordinary ground value can occur. A configuration represents the set of states that can be obtained by replacing configuration variables with all possible values. Each configuration variable is identified by a unique integer index and carries the type of values it stands for: either one of the Java primitive types, or the reference type along with a class name and some additional information, or the string type. A configuration variable can carry a restriction on the set of values. The configuration variables become the local variables of the residual program.

In the Java virtual machine, a program state consists of *global variables* (`static` fields of Java classes), a representation of *threads* and a *heap*.

In the Java Supercompiler, non-`final` global variables are not represented in a configuration, since at supercompilation time they are considered unknown and no information about them is kept. The values of `final static` fields are evaluated only once at the initialization stage, thus one copy of them is kept for all configurations. As the current version of JScp does not specialize multi-threaded code, the configuration contains only one thread now.

The definition of a *configuration* in the current JScp is as follows:

- a *configuration* is a triple (*thread*, *restrictions*, *heap*);
- a *thread* is a *call stack*, a sequence of *frames*;

- a *frame* is a triple (*local environment*, *evaluation stack*, *program point*);
- a *local environment* is a mapping of *local variables* to *configuration values*;
- an *evaluation stack* is a sequence of *configuration values*;
- the representation of a *program point* does not matter. It is sufficient to assume it allows us to resume supercompilation from the point;
- a *configuration value* is either a *ground value*, or a *configuration variable*;
- *restrictions* are a mapping *Restr* of configuration variables to predicates on their values. If a configuration variable v is not bound by the mapping, $Restr(v) = \lambda x.\mathtt{true}$. In the current version of JScp only restrictions of form $Restr(v) = \lambda x.(x \geq n)$, where n is an integer, on variables of the integral types of the Java language are implemented;
- we leave the notion of a *heap* unspecified here, since this paper does not deal with supercompilation of programs with objects.

The following three operations on configurations are used in JScp.

Comparison. of configurations for inclusion represented by a substitution: we consider $C_1 \subseteq C_2$ if there exist a substitution δ that binds configuration values to configuration variables such that $C_1 = \delta C_2$. Substitutions respect types and restrictions.

Generalization of configurations: a configuration G is the most specific *generalization of two configurations* C_1 and C_2 if $C_1 \subseteq G$ and $C_2 \subseteq G$ and for every G' satisfying this property, $G \subseteq G'$.

Homeomorphic embedding, a well-quasi order used for termination of loop unrolling: $C_1 \trianglelefteq C_2$ if the call stacks of C_1 and C_2 have the same "shape" (the lengths, the program points and the sets of local variables are the same) and $x_1 \trianglelefteq x_2$ holds for all pairs of corresponding configuration values x_1 from C_1 and x_2 from C_2, where $\trianglelefteq$ is the least relation satisfying:

- $v_1 \trianglelefteq v_2$ for all configuration variables v_1 and v_2 of the same type. If the configuration variables have an integral type, their restrictions must embed as well, $Restr(v_1) \trianglelefteq Restr(v_2)$ (see below);
- $x_1 \trianglelefteq x_2$ for all values x_1 and x_2 of the String class unless this is switched off by the user;
- $x_1 \trianglelefteq x_2$ for all ground values x_1 and x_2 of the same floating type;
- $n_1 \trianglelefteq n_2$ for all ground values n_1 and n_2 of the same integral type such that $0 \leq k \leq n_1 \leq n_2$ or $0 \geq -k \geq n_1 \geq n_2$, where k is a user-specified parameter that influences the depth of supercompilation. For verification of the protocols [12] values $k = 0$ and $k = 1$ were used (due to observation by A. Nemytykh);
- embedding of restrictions: $r_1 \trianglelefteq r_2$ if $r_1 = \lambda x.\mathtt{true}$, or $0 \leq n_1 \leq n_2$, or $0 \geq n_1 \geq n_2$, where $r_1 = \lambda x.(x \geq n_1)$ and $r_2 = \lambda x.(x \geq n_2)$.

Supercompilation of a method starts with the *initial configuration* comprised of one call stack frame with the method parameters bound to fresh configuration variables.

Driving. In supercompilation, the process of partial execution is referred to as *driving.*

Driving of method invocations. In the current version of JScp method invocations are either inlined, or residualized. No specialized methods are generated as in other supercompilers [19,22,23,24,25] and partial evaluators [10]. Whether to inline or not is controlled by certain JScp options. In our experiments on verification all method invocations were inlined.

Driving of expressions and assignments. Driving of an expression with a current configuration yields the value of the expression, residual code, and a new configuration. Driving is similar to interpretation with the following distinction.

Each unary or binary operation is either evaluated, if there is sufficient information to produce a ground resulting value, or otherwise residualized with a fresh configuration variable v as its value in form of a local variable declaration of form $t\ v$ `=` e, where e is the expression representing residualized operation with the values of arguments substituted into it.

Integer operations $v{+}i$ and $v{-}i$, where i an integer constant, v a configuration variable with restriction $\lambda x.(x \geq n)$, are residualized in form $t\ v'$ `=` $v + i$ and $t\ v'$ `=` $v - i$, and a new configuration variable v' with a restriction of form $\lambda x.(x \geq n + i)$ or $\lambda x.(x \geq n - i)$ is added to the configuration.

Integer comparisons v `==` i, v `!=` i, v `<` i, v `<=` i, v `>` i, v `>=` i and their commutative counterparts, where i is an integer ground value, v a configuration variable with restriction $\lambda x.(x \geq n)$, evaluate to `true` or `false`, when this is clear from comparison $n > i$ or $n \geq i$.

Driving of conditional statements. Consider a source code `if (`c`)` a `else` b`;` d, where c is a conditional expression, statements a and b are branches, statements d a continuation executed on exit from the `if` statement.

If driving of c yields `true` or `false`, the respective branch a or b is used for further driving. Otherwise, let c' be the residual code of the expression c, a configuration variable v its value. Two configurations C_t and C_f corresponding to the `true` and `false` branches are produced by taking into account the last operation of c'. If it is x `==` x' or x `!=` x', where x is a configuration variable, the configuration corresponding to equality is *contracted* [23], that is, substitution $x \mapsto x'$ is applied to the configuration. If the last operation is x `>` x', x `>=` x', x' `<` x, or x' `<=` x, where x is a configuration variable of an integral type, x' another variable or nonnegative integer value, the restriction on x is refined with information from x', if possible. Then each of the branches a and b is supercompiled with the respective initial configuration C_t and C_f, producing residual code a' and b' with final configurations C_a and C_b.

Supercompilation of d proceeds either two times with the initial configurations C_a and C_b, or once with C_g being the generalization of C_a and C_b. The choice between the alternatives is made by the JScp user. For the task of protocol verification we used the more aggressive first one.

The residual code of the `if` statement is either c' `;` `if (`v`)` $\{a'$ `;` $d'_a\}$ `else` $\{b'$ `;` $d'_b\}$, or c' `;` `if (`v`)` $\{a'$ `;` $\alpha_a\}$ `else` $\{b'$ `;` $\alpha_b\}$ `;` d', where d'_a, d'_b, and d' are residual codes of d from C_a, C_b, and C_g respectively, α_a and α_b are assignments that encode in Java the substitutions δ_a and δ_b that emerged from the generalization.

The `switch` statement is supercompiled analogously.

Configuration analysis of loop statements. Proper configuration analysis is performed only for loops in the current JScp. All kinds of loops in Java are reducible to a loop of form `L: while (true)` b, where b is a loop body statement.

Four kinds of exits are possible from the source and residual code of a loop body: `throw`, `return`, `break` and `continue`. The first three kinds are terminal nodes from the viewpoint of supercompilation of the loop statement. A `throw` statement is just residualized and no more actions are taken on that branch.A `return` statement is reduced to a `break` with a label of an appropriate enclosing statement. Processing of `break`s and `continue`s to a level higher than the loop statement is postponed until the corresponding level is reached. Statements `break L` are exits from the residual code of the loop statement. Residual statements `continue L` along with their configurations are subject to further configuration analysis.

Let a loop statement `L: while (true)` b be supercompiled with an initial configuration C_0. First, the loop body b is supercompiled with C_0 producing residual code b_0 and the list of statements `continue L` with configurations C_i, $i \in [1..n_0]$. For those C_i that $C_i \subseteq C_0$, $C_i = \delta_i C_0$, the `continue` statements are residualized in form α_i `; continue L`, where α_i are assignments encoding the substitution δ_i.

The remaining configurations C_i, $C_i \not\subseteq C_0$, comprise a current set $Cont$ of to-be-supercompiled `continue` statements. They are points of further loop unrolling: the loop body b is supercompiled with each $C \in Cont$ and the residual code is analyzed in the same way as for C_0.

This process is repeated and a residual code in form of a tree consisting of residual loop bodies supercompiled with various initial configurations is built. Each new configuration C_i on a leaf of an unfinished tree is checked for looping-back to all of the initial configurations of the residual loop bodies on the path from C_0 to this leaf.The process terminates when the set Cont is empty. However this does not happen in general case.

Generalization and termination. The most popular termination criterion [19,22,25] is based on the well-quasi-ordering. Before supercompiling the loop body with a next configuration C_i, the configuration is compared for homeomorphic embedding $\trianglelefteq$ (described above) with all of the previous initial configurations of the residual loop bodies on the path to it from C_0. If such C_j that $C_j \trianglelefteq C_i$ is found, C_j is generalized with C_i obtaining a configuration G, $C_j \subseteq G$. Then the residual subtree below C_j is erased, a sequence of assignments corresponding to the substitution δ that reduces C_j to G, $C_j = \delta G$, is inserted into the point of C_j, and supercompilation is repeated from the configuration G. This process terminates due to that there can be only a finite number of generalizations for each configuration and that our homeomorphic embedding $\trianglelefteq$ is well-quasi-order.

3 Application to Verification of Cache-Coherence Protocols

A. Lisitsa and A. Nemytykh [16,17,18] have found a nice class of applications solvable by supercompilation. They used the Refal Supercompiler SCP4 developed by A. Nemytkh and V. Turchin [19] and encoded in the functional language Refal the protocol models from Web site [6] developed by G. Delzanno [7]. The code and the results of supercompilation may be found on Web site [17].

Here we demonstrate this method of verification with the use of Java and the Java supercompiler JScp. The protocol models in Java and the results of supercompilation are collected on Web site [12]. The Java code of the models is rather close to the code in the domain-specific language HyTech used in [6].

For the description of the G. Delzanno's approach to the modeling of cache-coherence protocols, see his papers, e.g. [7]. Just the structure of the Java code of models is sufficient for explanation of the use of JScp.The models from [12] match the following pattern. It is commented in more detail in [13] together with a sample model of the MOESI cache-coherence protocol.

```
class model-class-name extends ProtocolModel {
  boolean runModel(int[] actions, int[] pars) throws ModelException {
    int state-var-1 = initial-value-1-or-pars[0]; ...
    require(precondition);
    for (int i = 0; i < actions.length; i++) {
      switch (actions[i]) {
        case 1: require(condition-for-action-1);
          computation-of-next-state; break;
        ...
        default: require(false);
    } }
    if (condition-for-unsafe-state-1) return false; ...
    return true;
  }
  void require(boolean b) throws ModelException {
    if (!b) throw new ModelException();
} }
```

To try to prove the correctness of a model we supercompile the method `runModel` and observe the residual code. If all `return` statements has form `return true`, we conclude the model can never reach an "unsafe" state, a state where the post-condition returns `false`.

4 Conclusion

We demonstrated application of the Java Supercompiler to verification of models belonging to the class of *counter systems*. There are a lot of works on decidability of various properties of these systems including reachability, to which verification reduces, and development of model-checkers for them. An overview can be found in some of the latest papers, e.g., [1]. As compared to these methods, supercompilation can be considered as generalization of forward analysis. The notion of *acceleration* in forward analysis of counter systems corresponds to that of *generalization* in supercompilation. Termination strategies based of well-quasi-orderings are close as well. The Java Supercompiler being a universal program specialization tool for a common object-oriented language is not as efficient and scalable as special-purpose tools and solves less problems from this class. However, its universality is an advantage for the ordinary user, allowing for combing program specialization tasks with verification of program from wider classes.

Supercompilation of the imperative subset of the Java language is worth comparing with works aimed at practical partial evaluation of imperative [3,5] and object-oriented languages [4,15,21]. The main distinctive feature of supercompilation is the explicit notion of a configuration with configuration variables and operations on configurations. This allows for more sophisticated analysis and transformation of programs, which is essential for program verification.

Acknowledgements. The development of the Java supercompiler would not be possible without collaboration with many people. The project was started together with Valentin Turchin and Larry Witte to whom the author is greatly indebted. Special thanks are due to the developers of various parts of the JScp system Arkady Klimov and Artem Shvorin. We are very grateful to our partners Ben Goertzel and Yuri Mostovoy: without their help and support such a complex project as JScp could not be done. It was a pleasure to collaborate with Andrei Nemytykh on application of supercompilation to program verification.

References

1. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: acceleration from theory to practice. International Journal on Software Tools for Technology Transfer 10(5), 401–424 (2008)
2. Broy, M., Zamulin, A.V. (eds.): PSI 2003. LNCS, vol. 2890. Springer, Heidelberg (2004)
3. Bulyonkov, M.A., Kochetov, D.V.: Practical aspects of specialization of algol-like programs. In: Danvy, O., Thiemann, P., Glück, R. (eds.) Dagstuhl Seminar 1996. LNCS, vol. 1110, pp. 17–32. Springer, Heidelberg (1996)
4. Chepovsky, A.M., Klimov, And.V., Klimov, Ark.V., Klimov, Y.A., Mishchenko, A.S., Romanenko, S.A., Skorobogatov, S.Y.: Partial evaluation for common intermediate language. In: Broy, Zamulin (eds.) [2], pp. 171–177
5. Consel, C., Lawall, J.L., Le Meur, A.-F.: A tour of Tempo: a program specializer for the C language. Sci. Comput. Program. 52, 341–370 (2004)
6. Delzanno, G.: Automatic Verification of Cache Coherence Protocols via Infinite-state Constraint-based Model Checking, `http://www.disi.unige.it/person/DelzannoG/protocol.html`
7. Delzanno, G.: Constraint-based verification of parameterized cache coherence protocols. Formal Methods in System Design 23(3), 257–301 (2003)
8. Ershov, A.P.: Mixed computation: potential applications and problems for study. Theoretical Computer Science 18, 41–67 (1982)

9. Goertzel, B., Klimov, A.V., Klimov, A.V.: Supercompiling Java Programs, white paper (2002), http://www.supercompilers.com/white_paper.shtml
10. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Englewood Cliffs (1993)
11. Klimov, And.V.: An approach to supercompilation for object-oriented languages: the Java Supercompiler case study. In: Nemytykh [20], pp. 43–53 (2008), http://meta2008.pereslavl.ru/accepted-papers/paper-info-4.html
12. Klimov, And.V.: JVer Project: Verification of Java programs by Java Supercompiler (2008), http://pat.keldysh.ru/jver/
13. Klimov, And.V.: A Java Supercompiler and its application to verification of cache-coherence protocols. In: Perspectives of Systems Informatics (Proc. 7th International Andrei Ershov Memorial Conference, PSI 2009), Novosibirsk, Russia, June 15-19, pp. 141–149. Ershov Institute of Informatics Systems (2009)
14. Klimov, And.V., Klimov, Ark.V., Shvorin, A.B.: The Java Supercompiler Project, http://www.supercompilers.ru
15. Klimov, Y.A.: An approach to polyvariant binding time analysis for a stack-based language. In: Nemytykh [20], pp. 78–84, http://meta2008.pereslavl.ru/accepted-papers/paper-info-6.html
16. Lisitsa, A.P., Nemytykh, A.P.: Towards verification via supercompilation. In: COMPSAC (2), pp. 9–10. IEEE Computer Society, Los Alamitos (2005)
17. Lisitsa, A.P., Nemytykh, A.P.: Experiments on verification via supercompilation (2007), http://refal.botik.ru/protocols/
18. Lisitsa, A.P., Nemytykh, A.P.: Reachability analysis in verification via supercompilation. Int. J. Found. Comput. Sci. 19(4), 953–969 (2008)
19. Nemytykh, A.P.: The supercompiler SCP4: General structure. In: Broy, Zamulin (eds.) [2], pp. 162–170
20. Nemytykh, A.P. (ed.): Proceedings of the First International Workshop on Metacomputation in Russia, July 2-5, 2008. Ailamazyan University of Pereslavl, Pereslavl-Zalessky (2008)
21. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for Java. ACM Trans. Program. Lang. Syst. 25(4), 452–499 (2003)
22. Sørensen, M.H., Glück, R.: An algorithm of generalization in positive supercompilation. In: Lloyd, J.W. (ed.) International Logic Programming Symposium, Portland, Oregon, December 4-7, pp. 465–479. MIT Press, Cambridge (1995)
23. Turchin, V.F.: The concept of a supercompiler. Transactions on Programming Languages and Systems 8(3), 292–325 (1986)
24. Turchin, V.F.: The algorithm of generalization in the supercompiler. In: Bjørner, D., Ershov, A.P., Jones, N.D. (eds.) Partial Evaluation and Mixed Computation, pp. 531–549. North-Holland, Amsterdam (1988)
25. Turchin, V.F.: Supercompilation: techniques and results. In: Bjorner, D., Broy, M., Pottosin, I.V. (eds.) PSI 1996. LNCS, vol. 1181, pp. 227–248. Springer, Heidelberg (1996)