An Approach to Supercompilation for Object-oriented Languages: the Java Supercompiler Case Study

Andrei V. Klimov[*]

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

Abstract. An extension of Turchin's supercompilation from functional to object-oriented languages as it is implemented in the current version of a Java supercompiler (JScp) is reviewed. There are two novelties: first, the construction of the specialized code of operations on objects is separated into two stages—residualization of all operations on objects during supercompilation proper and elimination of redundant code in post-processing; and second, limited configuration analysis, which processes each Java control statement one by one using width-first unfolding of a process graph, is used.
The construction of JScp is based on the principle of user control of the process of supercompilation rather than building a black-box automatic supercompiler. The rationale for this decision is discussed.
Keywords: specialization, supercompilation, object-oriented languages.

1 Introduction

Turchin's supercompilation [15] and related metacomputation technologies—partial evaluation, deforestation, mixed computation, etc.—for program specialization, fusion, slicing, inversion, etc., although being under development for more than three decades, are still in a state of infancy from the practical viewpoint. One may ask, why is it taking so long?

One evident reason is that time is always needed for a method to become mature enough to be embedded in tools and systems and used by rank-and-file programmers. However, from our viewpoint, there are essential reasons that have not allowed things to go this way quickly.

Supercompilation belongs to a new kind of program transformation technology which oversteps the limits of *black-box program optimization* used in widespread optimizing compilers. When a method is built in a subsystem that is almost invisible from outside, users accept it and it is put into practice easily.

[*] Supported by Russian Foundation for Basic Research projects 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.

To be “invisible”, a method must work quickly—preferably in linear time on the size of code. The commercial compiler developers consider this requirement essential. However, such limitation does not allow for unbounded evolution of the intelligence of program transformers.

In the early years of supercompilation development there was a dream of fully automatic supercompilers that could kind of “solve all problems” (note this is similar to the dream and belief in strong artificial intelligence in the same decades). But now, despite the evident progress in supercompilation and other metacomputation methods, we should adopt another viewpoint, another paradigm. Unbounded evolution is possible only in human-machine systems, the human performing the role of a metasystem [14].

Often the human control is considered as an interim measure with the goal to fully automate the control later and to commit it to the machine. It is indeed a good approach. In particular, it simplifies the first steps of system development by avoiding complex and possibly unsolvable problems in early stages. The only difference we argue for is that full automation should not be the ultimate goal. The goal must be careful division of labor between machine and human, putting at the machine level what it can do better, and on the human level what he can do better, and permanent movement of activities from human to machine, preserving the (meta) role of the human.

Such an approach requires development of new specific human-machine interfaces, and redevelopment of program transformation methods in such a way that they are comprehensible and controllable by the user.

Based on these considerations, from the very beginning, the Java supercompiler (JScp) [4,6] has been developing as a *user-controlled* system rather than an automatic supercompiler.

We have not yet constructed JScp as a convenient system for a user with an appropriate graphical user interface (GUI). At the current stage, this principle influenced the development in such a way that we were not afraid of introducing options in all places of supercompilation algorithms where there are degrees of freedom. Now the options are typed in a separate *advice file* which is an additional input to JScp. It is not easy to use JScp now. This is the main reason why we cannot suggest it to ordinary users. In future, a special GUI will have to be developed. This work has been just initiated and it is too early to discuss this topic in this paper in more detail.

Concluding the introductory part, I would like to share a strong impression based on experiments with the Java supercompiler that from the beginning of the Millennium we have practically no limitations from the hardware side for performing research and experiments in the area of metacomputation. The situation is quite opposite to what we had in the previous decades. Now we see that this was an objective reason for the slowness of the development of the methods. The few megabytes of memory that were not available in the 70s and were enough in the mid to late 80s to achieve self-application of specializers based on the method of partial evaluation [5,11], were totally insufficient to do what we do now. On the other hand, the modern gigahertzes and gigabytes give us an impressive freedom of experimenting, and in the nearest future we will observe the burst of results in our area.

2 Design Decisions of Java Supercompilation

2.1 User Control

As it was mentioned above, the Java supercompiler (JScp) has been developing as a *user-controlled* system rather than an automatic supercompiler. Let us turn to more tangible reasons for the decision:

– The project was initiated more than a decade ago, when research supercompilers had not yet shown themselves to be practical tools, while our goal was ambitious: to attack a practical language Java. There was not enough confidence in the possibility of automatic supercompilation at all, since experimental supercompilers (first of all, the V.F. Turchin's one [17,18]) were still weak.
– The main foreseeable problem was, is, and will be, *scalability* of the methods to large industrial code. Supercompilation as well as other relative methods have exponential (and perhaps even more) complexity. The user control is an effective (and perhaps, the only practicable) method to beat the exponential complexity down.
– The JScp project was a venture into supercompilation of the new object-oriented world, and a lot of experimenting to test and tune the methods was required.

Since then A.P. Nemytykh continued and has completed the development of the V.F. Turchin's supercompiler for the functional language Refal [10] and its practical usage has begun [9]. To a large extent, it may be considered as an automatic supercompiler. It solves a reasonable class of problems in almost automatic mode.[1] Nevertheless the mentioned uncertainties and risks remain, and we continue considering user-controlled rather than black-box supercompilers the high road of their development.

2.2 First Milestone

Construction of a supercompiler for such a cumbersome language as Java should be a stepwise process. Our first goal was to find such a subset of the supercompilation method "wheels" that is sufficient to implement the first supercompiler that has practical sense, kind of first "milestone". After it is achieved, we can get feedback from experiments with realistic code, and start the development of

[1] The combination of particular features implemented in it—negative information propagation at the level of driving and combination of V.F. Turchin's stack "whistle" [16] and the "whistle" [13,17] based on Kruskal's homeomorphism embedding [8]—have proven themselves to be highly successful.

convenient means to control the supercompiler by the user (which is a new area of research). We consider the JScp project being just at this stage.

In general outline, the current version of JScp (downloadable from the project site [6]) implements the following features:

- quite complete *driving*, which is, in particular, capable of rigorous specialization of operations on mutable objects. We observed that underdevelopments in driving noticeably reduce the depth of specialization. Nevertheless, some well-studied features of driving are not implemented yet: there is no negative information propagation and contractions after a test for equality are performed only for primitive data (and this feature is switched off by default). We observed these features are not as important than plain positive information propagation;
- limited *configuration analysis*. We gave up implementing the traditional configuration analysis that traversed all processes by driving and performed the operations on configurations to compare them, to loop-back, to generalize, to split, and thus constructed the residual graph. The main reason for this decision is that it is too monolithic: nearly a dozen of Java control statements must be fused into the algorithm of unfolding and folding the graph of configurations. For the first version of JScp we have made another decision where each control statement is driven, analyzed and residualized separately. This entailed the change from *depth-first* traversal of processes and configurations to *width-first*: for each control statement, its graph of configurations until the end of the statement is recursively built from the graphs of nested statements. This allowed us to elaborate the subtleties of each control statement one by one. Otherwise we did not manage the qualitative complexity of Java notions in the first version of JScp.

Thus there are two main differences in the method of supercompilation of Java implemented in the current version of JScp from the traditional supercompilation of functional languages:

- driving of operations on mutable objects (discussed in more detail in the next section). This is applicable to all object-oriented languages; and
- new method of configuration analysis of Java control statements by width-first unfolding of the graph of configurations and recursive constriction of residual code from the residual code of nested statements. (This is not further discussed in this paper.) This method is applicable not only to object-oriented languages, but to all imperative languages with a sophisticated set of control statements.

3 Driving of Operations on Objects

The most notable distinction of supercompilation in JScp is driving of objects.

An important feature of functional languages which is not preserved in object-oriented ones, and on which supercompilers and partial evaluators rely, is that values partially known at specialization time are easily residualized ("lifted"): a representation of a value (possibly with configuration variables) in a configuration can always be compiled into code producing the value at run-time, each execution of the code or its copies producing equal values.

Objects do not possess this property. It is no problem to represent the result of construction of an object as a result of driving of an instance creation expression `new` *C*`(`*arguments*`)`, where *C* is a class name, and to store it in the heap part of a configuration. It is not a problem to perform all operations on the representation of the object at supercompilation time. But it is impossible to generate the code that reconstructs the object at run-time. One of the reasons is that this code would generate different instances each time it is executed.

In "off-line" partial evaluators for object-oriented languages the preliminary binding time analysis supplies each instance creation expression with an annotation telling what to do: either to residualize the `new` expression, or not to residualize it and instead possibly residualize some of its fields as local variables. The necessity to take this decision in advance, when the values are completely unknown, restricts the depth of specialization. The preliminary analysis gives approximate information about the future of the objects.

3.1 Residualization of Operations on Objects

In "on-line" supercompilation, when driving meets an instance creation expression `new` *C*`(...)` it does not know whether the new object will be needed at run-time, or only some information from its fields. Hence, it is forced to always residualize it. The representation of the object is kept in configurations in order to perform operations on it at supercompilation time. If all information about the object is known then all operations will be performed by the supercompiler. Simultaneously all operations are residualized (except reading known values or configurations variables from fields) in order to create an object with the equivalent state at run-time.

In such a way, correct residual code is built but it contains a lot of redundant operations (see example in Fig. 3 below). Even local variables, which keep references to unused objects, may be unneeded. Often objects can be transformed to local variables that keep the values of part of fields needed at run-time.

Such transformation is performed in JScp by post-processing, which propagates information backwards from the points of use of objects to the points of their creation, from the future to the past.

3.2 Redundant Code Elimination by Post-processing

To eliminate redundant variables and code, the well-known methods from optimization compilers could be used. One might expect that an optimizing Java compiler can do this work and there is no need to implement this feature in JScp. However, all methods of redundant code elimination are approximate and address specific kinds of redundancy. The mainstream optimizing compilers are tuned for code written by humans or generated by preprocessors. It is unjustified to expect that they can find the redundant code produced by a supercompiler and perform expected transformations such as conversion of objects to local variables representing their fields.

Another reason why we have implemented a special-purpose post-processing analysis for redundant code elimination in JScp is that it is important to produce readable residual code—in particular, to support the user control that we argue for. (Compare the code in Fig. 3 and Fig. 4.)

The current version of post-processing analysis in JScp is a first approximation. It can be improved in future versions, but perhaps at the expense of time spent for analysis. The analysis is monovariant with respect to code, that is, all operations on reference variables are considered as an unordered set. One decision is made for each residual instance creation expression: whether to residualize it or not. If the instance is residualized, no fields are moved to local variables, although this may be beneficial in some branches of code.

Our experiments with supercompilation of realistic code show that such approximate analysis and transformation behaves rather well. The majority of redundant code is eliminated.

4 Example

Consider the famous A.P. Ershov's example of program specialization of a power function with respect to a known exponent. The only difference is that we use complex numbers represented by objects of class `Complex` (Fig. 1) instead of real numbers, in order to demonstrate how the two-stage residualization of objects works.

The program to be supercompiled is shown in Fig. 2. It consists of a general method `toPower`, which raises a complex number `x` to the power of an arbitrary nonnegative integer `n`, and a special method `toPower3`, which invokes the method `toPower` with `n = 3`.

The task for JScp is to supercompile method `toPower3`. In this case the JScp command line looks as follows:

```
jscp ComplexPower.java Complex.java -m toPower3 -aggr -invoke
```

where

– arguments `ComplexPower.java` and `Complex.java` are the source Java file names;
– option `-m toPower3` tells JScp to supercompile method `toPower3` from the first `.java` file;
– options `-aggr` and `-invoke` control supercompilation: the first one means using the standard set of "aggressive" options and the second one means unconditionally invoking (inlining) all method invocations at supercompilation time.

In Fig. 3 and Fig. 4 the actual output from the current version (0.1.99) of the Java supercompiler, which can be downloaded from the project site [6], is shown.

```
public final class Complex
{
    public final double re;
    public final double im;

    public Complex(double real, double imag)
    {
        re = real;
        im = imag;
    }
...
    public Complex times(Complex b)
    {
        Complex a = this;
        double real = a.re * b.re - a.im * b.im;
        double imag = a.re * b.im + a.im * b.re;
        return new Complex(real, imag);
    }
...
}
```

Fig. 1. A fragment of class Complex

```
public class ComplexPower
{
    public static Complex toPower(Complex x, int n)
    {
        Complex res = new Complex(1, 0);
        while (n != 0) {
            if (n % 2 == 1)
                { n=n-1; res = res.times(x); }
            else
                { n=n/2; x = x.times(x); }
            }
        return res;
        }

    public static Complex toPower3(Complex x)
    {
        return toPower(x, 3);
    }
}
```

Fig. 2. Source class ComplexPower and method toPower3 to be supercompiled

```
public static power.Complex toPower3(final power.Complex x_1)
{
  //{ 1 power.ComplexPower.toPower(power.Complex x_1, int 3)
  power.Complex res_2 = new power.Complex(1D, 0D);
  //{ 2 power.Complex res_2.times(power.Complex x_1)
  double re_5 = x_1.re;
  double im_6 = x_1.im;
  power.Complex res_7 = new power.Complex(re_5, im_6);
  //} 2 power.Complex res_2.times(power.Complex x_1)
  //{ 2 power.Complex x_1.times(power.Complex x_1)
  double double_12 = re_5 * re_5;
  double double_13 = im_6 * im_6;
  double real_14 = double_12 - double_13;
  double double_15 = re_5 * im_6;
  double double_16 = im_6 * re_5;
  double imag_17 = double_15 + double_16;
  power.Complex res_18 = new power.Complex(real_14, imag_17);
  //} 2 power.Complex x_1.times(power.Complex x_1)
  //{ 2 power.Complex res_7.times(power.Complex x_18)
  double double_23 = re_5 * real_14;
  double double_24 = im_6 * imag_17;
  double real_25 = double_23 - double_24;
  double double_26 = re_5 * imag_17;
  double double_27 = im_6 * real_14;
  double imag_28 = double_26 + double_27;
  power.Complex res_29 = new power.Complex(real_25, imag_28);
  //} 2 power.Complex res_7.times(power.Complex x_18)
  //} 1 power.ComplexPower.toPower(power.Complex x_1, int 3)
  return res_29;
}
```

Fig. 3. Residual method toPower3 before post-processing (underlined variables are redundant)

```
public static power.Complex toPower3(final power.Complex x_1)
{
  final double re_5 = x_1.re;
  final double im_6 = x_1.im;
  final double real_14 = re_5 * re_5 - im_6 * im_6;
  final double imag_17 = re_5 * im_6 + im_6 * re_5;
  return new power.Complex(re_5 * real_14 - im_6 * imag_17,
                           re_5 * imag_17 + im_6 * real_14);
}
```

Fig. 4. Residual method toPower3 after post-processing

Figure 3 contains the residual code before post-processing (which is output by option `-raw`). Notice the residual instance creation expressions, whose values are assigned to local variables with underlined names. The `new` expressions are redundant, since the underlined variables do not occur elsewhere. In this simple case the redundant variables are found even by the algorithm implemented in the Eclipse development platform. Its GUI shows this by similar underlining.

Figure 4 shows the final residual code. Post-processing also performs various equivalent transformations of code to make it more readable.

5 Conclusion and Related Work

The main contributions of our work on supercompilation of object-oriented languages are as follows:

- the method of supercompilation of code with mutable objects based on separation of the process into two stages: first, during driving and supercompilation proper, (almost) all operations on objects are residualized; second, thus obtained redundant code is eliminated by a specially developed post-processing;
- practical demonstration that a certain post-processing analysis is sufficient to eliminate the overwhelming majority of the redundant code;
- user-controlled configuration analysis based on width-first unfolding of a configuration graph rather than depth-first one used in existing supercompilers for functional languages.

To the best of our knowledge, this work is the first attempt to apply supercompilation-like methods to object-oriented languages. It goes without saying that it is based on previous works of various authors on supercompilation of functional languages, first of all on the works by V.F. Turchin. Before we have undertaken a venture of supercompilation of Java, it was very important for us to extract its essence from the gory details. The works [3] and [1] on simplification and clarification of basic notions of supercompilation were the most important for us to become optimistic.

The closest line of research is specialization of programs in object-oriented languages by partial evaluation. The main problem to be addressed is the same—evaluation of mutable objects at specialization time. However, the early work avoided this problem by restricting to immutable objects. Then, the method was extended to cover more and more parts of object-oriented notions. The most valuable works are that by U.P. Schultz et al. for Java [12] and a later one by Yu.A. Klimov et al. for the Common Intermediate Language (CIL) of the Microsoft.NET platform [2,7], which have extended the "polyvariance" of binding time analysis almost to the limit and allowed for all computations to be performed at specialization time when enough data is known.

As usual, supercompilation for object-oriented languages as an "on-line" technique is capable of performing deeper specialization than "off-line" partial evaluation. Experiments with partial evaluators and supercompilers show that application code and libraries often require to be refactored, but the amount of changes in the case of supercompilation are rather small and reasonable.

Concluding, we would like to say that the results of our development of the experimental Java supercompiler, the quality and even readability of the residual code, have exceeded our expectations, and hidden rocks turned out to be smaller than we were afraid of in advance.

6 Acknowledgments

The development of the Java supercompiler would not be possible without the whole of the team. The project was started together with Larry Witte and Valentin Turchin. The author is greatly indebted to the main 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 could not be accomplished.

The ideas and results of Java supercompilation were discussed at the seminar on Refal and metacomputation at Keldysh Institute. Special thanks to its most active participants: Sergei Abramov, Arkady Klimov, Yuri Klimov, Ilya Klyuchnikov, Andrei Nemytykh, Leonid Provorov, Igor Shchenkov, Sergei Romanenko, Anton Orlov, and others.

Comments by Geoff Hamilton on the draft of this paper were invaluable.

References

1. Sergei M. Abramov. *Metavychislenija i ikh prilozhenija (Metacomputation and its applications)*. Nauka, Moscow, 1995. (In Russian).
2. Andrei M. Chepovsky, Andrei V. Klimov, Arkady V. Klimov, Yuri A. Klimov, Andrei S. Mishchenko, Sergei A. Romanenko, and Sergei Yu. Skorobogatov. Partial evaluation for common intermediate language. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers*, volume 2890 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2003.
3. Robert Glück and Andrei V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filè, and G. Rauzy, editors, *Static Analysis Symposium. Proceedings*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 1993.
4. Ben Goertzel, Andrei V. Klimov, and Arkady V. Klimov. *Supercompiling Java Programs, white paper*, 2002. http://www.supercompilers.com/white_paper.shtml.
5. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
6. Andrei V. Klimov, Arkady V. Klimov, and Artem B. Shvorin. *The Java Supercompiler Project*. http://www.supercompilers.ru.
7. Yuri A. Klimov. Program specialization for object-oriented languages by partial evaluation: approaches and problems. Preprint 28, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 2008. (In Russian).

8. Joseph B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960.
9. Alexei Lisitsa and Andrei P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.
10. Andrei P. Nemytykh. *Superkompilyator SCP4: Obschaya struktura (The Supercompiler SCP4: General Structure).* URSS, Moscow, 2007. (In Russian).
11. Sergei A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.
12. Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for Java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.
13. Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *International Logic Programming Symposium, Portland, Oregon.* MIT Press, 1995. (To appear).
14. Valentin F. Turchin. *The Phenomenon of Science.* Columbia University Press, New York, 1977.
15. Valentin F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
16. Valentin F. Turchin. The algorithm of generalization in the supercompiler. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.
17. Valentin F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 481–509. Springer, 1996.
18. Valentin F. Turchin. Supercompilation: techniques and results. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, June 25-28, 1996, Proceedings*, volume 1181 of *Lecture Notes in Computer Science*, pages 227–248. Springer, 1996.