An Interactive Specializer Based on Partial Evaluation for a Java Subset

[1] *I. A. Adamovich <i.a.adamovich@gmail.com>*
[2] *And. V. Klimov <klimov@keldysh.ru>*
[1] *Ailamazyan Program Systems Institute of Russian Academy of Sciences, 4a Peter the First str., Veskovo, Yaroslavl region, 152021, Russia*
[2] *Keldysh Institute of Applied Mathematics of Russian Academy of Sciences, 4 Miusskaya sq., Moscow, 125047, Russia*

Abstract. Specialization is a program optimization approach that implies the use of a priori information about values of some variables. Specialization methods are being developed since 1970s (mixed computations, partial evaluation, supercompilation). However, it is surprising, that even after three decades, these promising methods have not been put into the wide programming practice. One may wonder: What is the reason? Our hypothesis is that the task of specialization requires much greater human involvement into the specialization process, the analysis of its results and conducting computer experiments than in the case of common program optimization in compilers. Hence, specializers should be embedded into integrated development environments (IDE) familiar to programmers and appropriate interactive tools should be developed. In this paper we provide a work-in-progress report on results of development of an interactive specializer based on partial evaluation for a subset of the Java programming language. The specializer has been implemented within the popular Eclipse IDE. Scenarios of the human-machine dialogue with the specializer and interactive tools to compose the specialization task and to control the process of specialization are under development. An example of application of the current version of the specializer is shown. The residual program runs several times faster than the source one.

Keywords: program analysis, program transformation, interactive program specialization, partial evaluation, object-oriented language, integrated development environment.

DOI: 10.15514/ISPRAS-2018-30(4)-2

For citation: Adamovich I.A., Klimov And.V. An Interactive Specializer Based on Partial Evaluation for a Java Subset. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 4, 2018. pp. 29-44. DOI: 10.15514/ISPRAS-2018-30(4)-2

[1] Supported by RFBR research project No. 18-37-00454 (contribution: development of interactive methods of partial evaluation, design of the architecture and implementation of the specializer, analysis of related works).
[2] Supported by RFBR research project No. 16-01-00813 (contribution: problem statement, design of methods based on the existing approaches, supervision, analysis of related works).

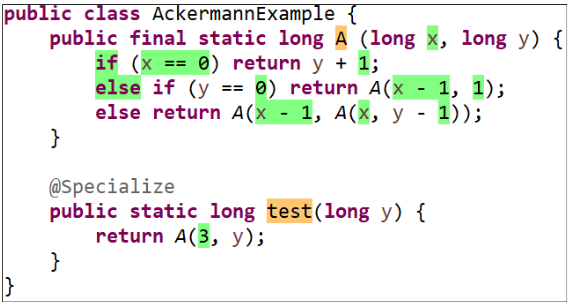
1. Introduction

The method of program specialization known as *partial evaluation* was invented more than 30 years ago along with the achievement of the famous result [1], [2] of evaluation of the First, Second and Third Futamura projections [3]–[5] for a tiny List subset. The first round of research was completed in early 1990s when the main textbook on partial evaluation had been published [2]. A lot of programming problems were found to be solved by program specialization (the most known being the generation of a compiler from an interpreter by the Second Futamura Projection) and the emergence of a new class of program development tools based on specialization were expected. Some other program specialization techniques, e.g., *supercompilation* [6], [7], has been developed in parallel as well. However, it is surprising that even after three decades these promising methods have not been put into the wide programming practice. One may wonder: What is the reason?

Our hypothesis is that the main expectation that governed the development of specializers was wrong. The developers of these methods hoped that specializers could work in fully automatic mode and they just needed to invent some finitely many features and improvements that solve the problem, after which “the great goal” would be achieved and happy programmers started using the new tools. They expected that specializers could work in the similar “black-box mode” as optimizing compilers. However this did not happen. The time and space complexity of the program transformations that were necessary for specialization, turned out to be much higher than the complexity of program optimizations that can be used as “black boxes” with short and predictable run time and consumed memory.

We argue that automatic methods of program optimization have reached certain inherent limits. In order to develop and use more powerful tools, we must give up the expectations that the program analysis and transformation systems will operate in automatic mode without human intervention. Program specializers possess too many degrees of freedom and choice, which cannot be resolved by the algorithms of their kind and, therefore, should use human help.

Based on this observation, we put forward the goal of construction of an interactive specializer embedded in a habitual integrated development environment (IDE) such as Eclipse [8]. Eclipse provides a rich open-source toolkit referred to as Java development tools (JDT) [9], which allows a developer to deal only with essential tasks of analysis, visualization and transformation of Java code. Adequate human-machine dialogue tools to control the specializer and deal with the results of specialization are to be developed. We would like to emphasize that there is strict separation of concerns between the machine and the human: the specializer guaranties the functional equivalence of program transformation and the user is responsible for the control of the specializer in such a way that it produces the code that satisfied user’s goals and needs (which the machine does not know).



```
public class AckermannExample {
    public final static long A (long x, long y) {
        if (x == 0) return y + 1;
        else if (y == 0) return A(x - 1, 1);
        else return A(x - 1, A(x, y - 1));
    }

    @Specialize
    public static long test(long y) {
        return A(3, y);
    }
}
```

Fig.1. Source code of Ackermann function

We think that partial evaluation is better suited than other specialization methods (like supercompilation) for human-machine dialogue organized in such a way that the user comprehends what is happing in the specializer, receives valuable and interesting information about his code, is capable of adjusting the source code to be better specialized and controls the specializer. The reason is that the method of partial evaluation consists of two stages:

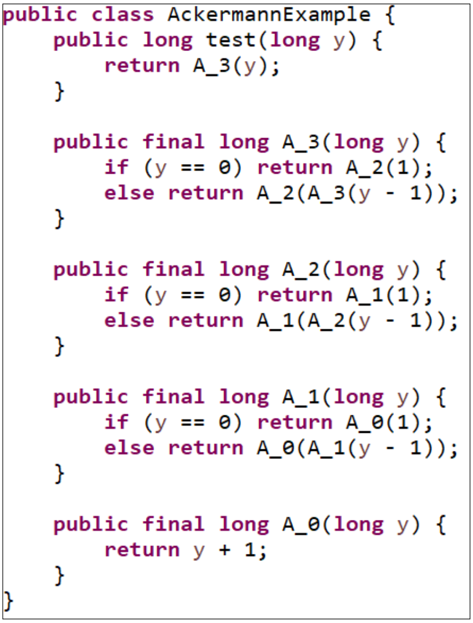
- *binding-time analysis* (BTA) of source code that selects the parts of the code that are to be evaluated at specialization time, and
- *residual program generation* (RPG) that follows the information supplied by BTA, performs specialization proper and produces the resulting code (referred to as *residual*).

A pleasant feature of BTA is that its result (called *BT annotation*) may be naturally shown on the source code by highlighting and due to such visualization the residual code is intuitively predictable. We hope that this will allow for easy adoption of specializers as new programming tools by rank-and-file programmers.

Terminological remark. In the theory of partial evaluation the parts of source code to be evaluated during specialization are called *static*. The other source code that is transferred to the residual program (*residualized*) is referred to as *dynamic*. The term *static* conflicts with the `static` modifier in Java and the term *dynamic* may be confused with the run-time notions. That is why we avoid using these words in the partial evaluation sense and use abbreviations `S` and `D` instead, e.g., `S`-annotation, `D`-annotation, `S`-code, `D`-code, `S`-part and `D`-part of a program.

The contributions of this paper are as follows.

- We show the first results of development of the Java specializer, where partially evaluated code is restricted to operations on primitive types.
- We demonstrate the work of the specializer by an example of specialization of the Ackermann function with respect to the first argument.
- We discuss some of the details of implementation in Eclipse and the methods and features to be implemented in future.



```
public class AckermannExample {
    public long test(long y) {
        return A_3(y);
    }

    public final long A_3(long y) {
        if (y == 0) return A_2(1);
        else return A_2(A_3(y - 1));
    }

    public final long A_2(long y) {
        if (y == 0) return A_1(1);
        else return A_1(A_2(y - 1));
    }

    public final long A_1(long y) {
        if (y == 0) return A_0(1);
        else return A_0(A_1(y - 1));
    }

    public final long A_0(long y) {
        return y + 1;
    }
}
```

Fig. 2. Residual code of Ackermann function

The outline of the paper is as follows. In Section 2 we present the basics of partial evaluation for Java by an example of specialization of the Ackermann function. In Section 3 a bird-eye view of the implementation of the specializer in the Eclipse IDE is presented. Section 4 contains a survey of related works in comparison with our specializer. In Section 5 we conclude.

2. Java Specialization by Example

Fig. 1 and 2 contain screenshots of the source and residual code of the Ackermann function made from the running specializer in Eclipse IDE.

The method `A` implements the Ackermann function and the method `test` invokes it with the first constant argument `3`. The Java annotation `@Specialize` at the method `test` specifies that it should be specialized, i.e., its body is to be replaced with the residual code and the specialized versions of the methods that it invokes are to be generated and added to the program. The names of the methods `A` and `test` in their headers are marked in orange in order to show that they are involved in BTA. The bodies of these methods are analyzed and annotated: green highlighting marks `S`-parts of code. (You see gray highlighting in fig. 1 if you read this paper in a monochrome print).

2.1. Binding-Time Analysis

The BTA algorithm for variables and operations of primitive types is rather straightforward. First, all constants are annotated with `S`. Then recursively: a subexpression containing only `S`-parts becomes `S`; a local variable declaration and an assignment with `S` right-hand sides become `S`; a method parameter that correspond to `S` arguments at all points of invocation becomes `S`; in case of conflict of several invocations of the same method the conflicting parameter becomes `D`; a conflict on several assignments to a local variable turns it to `D` as well; an `if` statement with the `S` conditional expression is annotated with `S` regardless of the annotation of its branches (this means that `if-else` will disappear while one of the branches will be residualized); other control statements are analyzed and annotated similarly. When the recursion reaches the fixed point, the remaining parts of code are annotated with `D`. `D`-parts are not highlighted in Figure 1.

This mode of operation of BTA, when each code fragment gets univocal annotation `S` or `D`, is referred to as *monovariant*. The more general mode when several versions of annotation are allowed is called *polyvariant*. The current version of BTA is monovariant. In future we plan to implement polyvariant BTA for classes and reference types according the theory developed in [10]–[18].

Monovariant BTA on primitive types can be defined formally as abstract interpretation on a lattice with 3 elements: *undefined* < `S` < `D`.

As an illustration of monovariance, notice that in figure 1 method `A` is invoked 3 times in the source code, one of which has both `S` arguments, another 2 invocations have the first `S` argument and the second one is `D`. The first invocation is processed in the same way as the other two with the second `S` argument assigned to the `D` formal parameter.

2.2. Residual Program Generation

At the generation stage, partial evaluation starts from the method with the `@Specialize` annotation and recursively visits all invoked methods in turn. Notice that, since all statements and methods with side effects are considered `D` and hence are residualized rather than executed at specialization time, the order of specialization of methods does not matter. For each of the specialized methods, several residual versions can be produced — one for each combination of values of `S` arguments. They got different names of the form (in the current version): *source-name_number*. They have only those parameters that correspond to `D` parameters in the source code.

The current version of the specializer can loop forever if infinitely many values of `S` arguments are generated. The production version of the specializer should contain special debugging means to gracefully leave such situations. This is our future work. In Figure 2 there are 4 versions of residual method `A` corresponding to values `0`, `1`, `2`, `3` of its first argument. Notice that because of monovariance the invocations

`A_2(1)`, `A_1(1)`, and `A_0(1)` have not being evaluated, since the constant `1` correspond to the `D` parameter of method `A`.

2.3. Running Source and Residual Programs

We have chosen this example for presentation, since it demonstrates all main features of the current version of the specializer. We did not expect a significant speed-up as it seemed that asymptotically the number of method invocations was almost the same and the invocations were the most expensive operations in this example. Thus we were very surprised when the speed-up was about 3 times.

The obtained acceleration can be explained by several reasons. First, calculation showed that the specialized version performs 1.86 times less Java byte code instructions. Second and more important, it is natural to suppose that the JIT compiler in JVM performs inlining of those specialized method that are simpler and more compact than in the source code.

This example illustrates the principle, which we observed many times in experiments with various specializers: a specializer does not replace the classic optimizing compilers. Rather, we observe “composition” of optimizations by a specializer and a low-level optimizing compiler and hence multiplication of speed-ups. Residual code produced by specializers is more amendable for classic optimizations than code written by a human being. We may conclude that specialization opens up additional opportunities for program optimization.

3. Architecture of Specializer

The specializer has been implemented in the Eclipse development environment (IDE) [8]. The IDE has open source code and provides points and tools to extend it.

The basis for Eclipse extension is the concept of a plug-in. Each plug-in is an archive JAR file containing a so-called manifest, a set of files describing the dependencies of the plug-in and the possibility of its extension (extension points). Other plug-ins can add their functionality to these extension points. For example, one might want to add his toolbar extensions to an already implemented toolbar plug-in.

A small tool is usually implemented as a one plug-in, while a large one is often provided as a set of plug-ins. Our specializer is implemented as three Eclipse plug-ins.

The specializer consists of the following plug-ins:

- a plug-in SpecCore is the core of the specializer, which implements its main functionality;
- a plug-in SpecMarkers is responsible for text highlighting in the Eclipse editor in accordance with the annotation produced by the SpecCore plug-in;
- a plug-in SpecMenus implements interactions with various menus (including context menus) and toolbars to provide a user-friendly interface.

The SpecCore implements the binding-time analysis (BTA) and the generation of a residual program. When analyzing the source program the plug-in SpecCore uses the abstract syntax tree (AST) built by the Eclipse Java development tools (JDT).

JDT is a set of plug-ins that provides us with an easy way to manipulate Java source code.

The second of the three plug-ins that form the specializer is the SpecMarkers plug-in. It is responsible for highlighting the source code, which allows the programmer to see which parts of the program are evaluated at specialization time and which are residualized. This helps him to understand how to change the code to provide better specialization.

The last part of the specializer is the SpecMenus plug-in. This plug-in uses the extension points of other plug-ins to add the necessary elements to some menus. It adds two new buttons to the main toolbar of Eclipse: Enable/Disable the highlighting and the “Generate optimized Java files” button. Also this plug-in adds items to the context menu of the Project Explorer and Package Explorer views.

4. Related Work and Comparison

A lot of works are devoted to partial evaluation for functional languages. The book [2] summarizes the first wave of development of this method.

Later on, research into partial evaluation for imperative “Algol-like” languages [19], [20] and C [21] was performed. In early 1990's, the first (to our knowledge) specializer for C was developed, called C-MIX [21], [22]. Chapter 11 of the book [2] contains its detailed presentation. C-MIX specializes a program in three stages.

The first stage is the analysis of references. For each reference variable, a set of the variables that it could refer to is built. If the analysis finds that several reference variables can refer to the same memory, they are labeled identically. The second stage is the construction of a binding-time annotation of the source code. References to the same memory area are annotated identically. In case of conflicts, the annotation is reduced to `D` as usual. The third stage is the generation of the residual program.

Specialization of reference types in Java can be similar to elaboration of pointers in C-MIX. However, Java stricter typing and managed run-time can be leveraged for deeper specialization. The current version of our specializer annotates all reference variables `D` and, therefore, they are left unchanged. Our future work is to add the binding-time analysis of reference types. Unlike C-MIX, we expect that our specializer will still work in two stages — without the reference analysis phase.

Further development of ideas of C-MIX led to the creation of a specializer of programs written in C, called Tempo [23], [24]. This specializer is much like C-MIX.

The next important step was the development of the first specializer for an object-oriented language — JSpec for Java [25]. JSpec uses the Harissa compiler [26] to translate the Java program into C. Then the Tempo specializer mentioned above transforms the program. The obtained C-representation of a specialized Java program is mapped back into Java using the Assirah translator [25]. Finally, the AspectJ tool weaves the specialized program with the source program to get the executable Java bytecode. The main limitation of JSpec is that it is capable of partially evaluating only immutable classes and objects, while mutable objects are always residualized. Our goal is to waive this restriction.

The most advanced (to our knowledge) partial evaluation method for object-oriented languages like C# and Java has been developed in CILPE [10]–[18], a partial evaluator for Common Intermediate Language (CIL), the bytecode of the Microsoft .NET Framework. It supports almost all of the basic constructs of object-oriented languages such as C# and Java. In CILPE, a new concept of a binding-time heap (BT heap) has been introduced. A BT heap is an abstract description of the state of a run-time heap, which allows us to separate reference type data into evaluated at specialization time and residualized ones and to avoid the use of the reference analysis stage as in C-MIX. As a result of specialization, some of the objects are no longer created in the residual program, and if necessary, local variables are used instead of object fields. We will base on the results of this research in our future work to implement BTA of classes and partial evaluation of objects.

A relatively new specializer of Java programs is Civet [27]. Civet is based on a so-called Hybrid Partial Evaluation (HPE) approach. Specialization in HPE is performed in *online* mode, i.e., in one pass, while the programmer can specify which parts of the program have `S`-annotation. On the contrary, in our specializer we choose the *offline* approach, i.e., the residual program is built at the stage of generation of the residual program after the completion of the binding-time analysis, where information about the `S`-parts of the program is collected automatically rather than specified by the user as in Civet[1].

PE-KeY [28] is a partial evaluator for Java programs based on the KeY verification system [29]. PE-Key works in two stages. At the first stage, the program is executed in a symbolic form with the application of a special set of rules. At the second stage, a residual program is synthesized, while the rules are applied in the opposite direction. The PE-KeY approach is similar to the classical offline specialization that our specializer uses: a specialized program is produced in two stages. However, in the first stage of PE-KeY, the program is executed symbolically, while our binding-time analysis performs an abstract interpretation of the program. In addition, due to limitations of the KeY verification system, PE-KeY does not support floating-point arithmetic, while our specializer supports.

JSpec, Civet, PE-Key deal with objects at specialization time, while the current version of our specializer annotates classes and variables of reference types with `D`

[1] For discussion of the features of and differences between online and offline partial evaluation see [2, Chapter 7].

and thus residualizes them unchanged. The extension of our specializer to partial evaluation of classes and objects is our future work.

The specializers considered above interact with the user through the command line, so it's extremely difficult to use them. In order for the specialization to be widely used, it is required to develop the methods of interaction with the user and to embed the specializer into an integrated development environment convenient for the programmer, what we are implementing in our specializer. This is a crucial difference.

We know about just one work on partial evaluation carried out in a practical setting – the GraalVM toolkit in Oracle Labs [30], [31]. The toolkit is designed for defining domain-specific languages by interpreters and, nevertheless, achieving high-performance by using a specializer. The first Futamura projection provides an opportunity for such acceleration (see [3], [4] and [2, Chapter 1.5.1]): given a program and an interpreter that executes the program, GraalVM specializes the interpreter with respect to a part of the given program and produces the machine code of this part. The resulting code is executed much faster than the original one in the interpreter. The main goal of GraalVM is to provide a technology similar to just-in-time (JIT) compilation for the developer of a programming language without the need to implement the complex machinery of JIT. The interpreter specialization in GraalVM is not automatic and uses prompts by the interpreter developer. This case of implementation of partial evaluation confirms that practical application of specialization requires guidance from the programmer. We conduct our research in the same direction: methods and tools are being developed to provide the programmer with information about program behavior under specialization and levers to control the partial evaluation processes.

5. Conclusion

In this paper we put forward the task of development of an interactive specializer.

We argue that the current stage of program specialization methods has reached certain limits because the previously implemented specializers do not imply the participation of the user in the process of specialization. Our specializer uses the offline partial evaluation approach, where the program transformation if performed in two stages — binding-time analysis (BTA) and residual program generation (RPG). We briefly described the architecture of our interactive specializer in the Eclipse development environment.

We illustrated the work of the specializer with the famous example of the Ackermann function and the result of its specialization with respect to its first argument. The specialized program runs several times (about three) faster than the original one.

We see the following directions for further development of the specializer:

- to develop and implement binding-time analysis and residual program generation for classes and objects;

- to implement interactive tools for composing a specialization task and controlling the process of binding-time analysis and residual program generation;
- to implement tools to visualize the correspondence between source and residual code;
- to demonstrate that a well-developed specializer can convert well-structured high-level human-oriented code, which can not be automatically parallelized, into code that can be parallelized by existing methods and tools;
- to prepare demo programs that benefit from specialization, for example, building a compiler from an interpreter;
- to generalize the binding-time analysis from monovariant to polyvariant;
- to develop an interactive tracer (similar to run-time debuggers) that allows the user to observe the analysis and generation processes in order to improve the behavior of his code under specialization.

Availability. The current version of our specializer is available at ftp://ftp.botik.ru/rented/iaadamovich/Specializer/.

Acknowledgment

We are grateful to our friends and colleagues Yuri Klimov, Arkady Klimov, Sergei Romanenko, Sergei Abramov for valuable advices on specialization methods in general and partial evaluation in particular and constructive feedback on the design of our specializer system.

References

[1]. Jones N.D., Sestoft P. and Søndergaard H. An experiment in partial evaluation: the generation of a compiler generator. Rewriting Techniques and Applications, Lecture Notes in Computer Science, J.-P. Jouannaud, (Ed.), vol. 202. Springer-Verlag, 1985, pp. 124–140

[2]. Jones N.D., Gomard C.K., and Sestoft P. Partial Evaluation and Automatic Program Generation. Prentice-Hall, 1993, 415 p. Available at: http://www.itu.dk/~sestoft/pebook/pebook.html, accessed 20.06.2018

[3]. Futamura Y. Partial evaluation of computation process — an approach to a compiler-compiler. Systems, Computers, Controls, vol. 2, no. 5, 1971, pp. 45–50

[4]. Futamura Y. Partial evaluation of computation process — an approach to a compiler-compiler. Higher-Order and Symbolic Computation, vol. 12, no. 4, Dec 1999, pp. 381–391. Updated and revised version of [3]. Available at: http://doi.org/10.1023/A:1010095604496, accessed 20.06.2018

[5]. Futamura Y. EL1 Partial Evaluator (Progress Report). Center for Research in Computing Technology, Harvard University, Tech. Rep., 1973. Available at: http://fi.ftmr.info/PE-Museum/EL1.PDF, accessed 20.06.2018

[6]. Turchin V.F. The concept of a supercompiler. ACM Transactions on Programming Languages and Systems, vol. 8, no. 3, 1986, pp. 292–325

[7]. Turchin V.F. Supercompilation: techniques and results. Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, June 25-28, 1996. Proceedings, Lecture Notes in Computer Science, D. Bjørner, M. Broy, and I.V. Pottosin, (Eds.), vol. 1181. Springer, 1996, pp. 227–248

[8]. Eclipse Foundation. Eclipse Integrated Development Environment (IDE). Available at: https://www.eclipse.org, accessed 20.06.2018

[9]. Eclipse Foundation. Eclipse Java development tools (JDT). Available at: https://www.eclipse.org/jdt, accessed 20.06.2018

[10]. Klimov Yu.A. An approach to polyvariant binding time analysis for a stack-based language. First International Workshop on Metacomputation in Russia, Proceedings. Pereslavl-Zalessky, Russia, July 2–5, 2008. Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008, pp. 78–84. Available at: http://meta2008.pereslavl.ru/accepted-papers/paper-info-6.html, accessed 20.06.2018

[11]. Klimov Yu.A. [Program specialization for object-oriented languages by partial evaluation: approaches and problems]. Preprinty` IPM im. M.V. Keldy`sha [Keldysh Institute Preprints], no. 12, 2008 (in Russian). Available at: http://library.keldysh.ru/preprint.asp?id=2008-12, accessed 20.06.2018

[12]. Klimov Yu.A. [Specializer CILPE: examples of object-oriented program specialization]. Preprinty` IPM im. M.V. Keldy`sha [Keldysh Institute Preprints], no. 30, 2008 (in Russian). Available at: http://library.keldysh.ru/preprint.asp?id=2008-30, accessed 20.06.2018

[13]. Klimov Yu.A. [SOOL: an object-oriented stacked-based language for specification and implementation of program specialization techniques]. Preprinty` IPM im. M.V. Keldy`sha [Keldysh Institute Preprints], no. 44, 2008 (in Russian). Available at: http://library.keldysh.ru/preprint.asp?id=2008-44, accessed 20.06.2018

[14]. Klimov Yu.A. [Specializer CILPE: binding time analysis]. Preprinty` IPM im. M.V. Keldy`sha [Keldysh Institute Preprints], no. 7, 2009 (in Russian). Available at: http://library.keldysh.ru/preprint.asp?id=2009-07, accessed 20.06.2018

[15]. Klimov Yu.A. [Specializer CILPE: residual program generation]. Preprinty` IPM im. M.V. Keldy`sha [Keldysh Institute Preprints], no. 8, 2009 (in Russian). Available at: http://library.keldysh.ru/preprint.asp?id=2009-08, accessed 20.06.2018

[16]. Klimov Yu.A. [Specializer CILPE: correctness proof]. Preprinty` IPM im. M.V. Keldy`sha [Keldysh Institute Preprints], no. 33, 2009, (in Russian). Available at: http://library.keldysh.ru/preprint.asp?id=2009-33, accssed 20.06.2018

[17]. Klimov Yu.A. [Specialization of programs in object-oriented languages by partial evaluation]. Ph.D. dissertation, Keldysh Institute of Applied Mathematics of RAS, Moscow, Russia, Nov 2009, 183 p. (in Russian). Available at: http://pat.keldysh.ru/~yura/publications/2009.10-Klimov-Disser-Specializacia_programm_na_ob'ektno-orientirovannyx_yazykah.pdf, accessed 20.06.2018

[18]. Klimov Yu.A. [Specializer CILPE: Partial evaluation for object-oriented languages]. Programmny`e sistemy`: teoriia i prilozheniia [Program Systems: Theory and Applications], no. 3(3), pp. 13–36, 2010 (in Russian). Available at: http://psta.psiras.ru/read/psta2010_3_13-36.pdf, accessed 20.06.2018

[19]. Bulyonkov M.A. and Kochetov D.V. Practical aspects of specialization of Algol-like programs. Dagstuhl Seminar on Partial Evaluation, Lecture Notes in Computer Science, O. Danvy, R. Gluck, and P. Thiemann, (Eds.), vol. 1110. Springer, 1996, pp. 17–32

[20]. Ershov A.P. and Itkin V.E. Correctness of mixed computation in Algol-like programs. MFCS, Lecture Notes in Computer Science, J. Gruska, (Ed.), vol. 53. Springer, 1977, pp. 59–77

[21]. Andersen L.O. Program analysis and specialization for the C programming language. Ph.D. dissertation, DIKU, University of Copenhagen, May 1994, (DIKU report 94/19)

[22]. Andersen L.O. Binding-time analysis and the taming of C pointers. Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '93). ACM, 1993, pp. 47-58. Available at: http://dx.doi.org/10.1145/154630.154636, accessed: 20.06.2018

[23]. Consel C., Lawall J.L., and Meur A.-F.L. A tour of Tempo: a program specializer for the C language. Sci. Comput. Program., vol. 52, no. 1-3, 2004, pp. 341–370

[24]. Meur A.L., Lawall J.L. and Consel C. Towards bridging the gap between programming languages and partial evaluation. Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM ’02), Portland, Oregon, USA, January 14-15, 2002, P. Thiemann, (Ed.). ACM, 2002, pp. 9–18. Available at: http://doi.acm.org/10.1145/503032.503033, accessed 20.06.2018

[25]. Schultz U.P., Lawall J.L. and Consel C. Automatic program specialization for Java. ACM Trans. Program. Lang. Syst., vol. 25, no. 4, 2003, pp. 452–499

[26]. Muller G., Moura B., Bellard F. and Consel C. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS), June 16-20, 1997, Portland, Oregon, USA, S. Vinoski, (Ed.). USENIX, 1997, pp. 1–20. Available at: http://www.usenix.org/publications/library/proceedings/coots97/muller.html, accessed 20.06.2018.

[27]. Shali A. and Cook W.R. Hybrid partial evaluation. SIGPLAN Not., vol. 46, no. 10, Oct. 2011, pp. 375–390. Available at: http://doi.acm.org/10.1145/2076021.2048098, accessed 20.06.2018.

[28]. Ji R. and Bubel R. PE-KeY: A partial evaluator for Java programs. Proceedings of the 9th International Conference on Integrated Formal Methods, IFM’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 283– 295. Available at: http://dx.doi.org/10.1007/978-3-642-30729-4_20, accessed 20.06.2018

[29]. Ahrendt W., Beckert B., Bubel R., Hahnle R., Schmitt P.H. and Ulbrich M., (Eds.). Deductive Software Verification – The KeY Book – From Theory to Practice. Lecture Notes in Computer Science. Springer, 2016, vol. 10001. Available at: https://doi.org/10.1007/978-3-319-49812-6, accessed 20.06.2018

[30]. Würthinger T., Wimmer C., Wöß A., Stadler L., Duboscq G., Humer C., Richards G., Simon D., and Wolczko M. One VM to rule them all. Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013. New York, NY, USA: ACM, 2013, pp. 187–204. Available at: http://doi.acm.org/10.1145/2509578.2509581, accessed 20.06.2018

[31]. Würthinger T., Wimmer C., Humer C., Wöß A., Stadler L., Seaton C., Duboscq G., Simon D., and Grimmer M. Practical partial evaluation for high-performance dynamic language runtimes. SIGPLAN Not., vol. 52, no. 6, Jun. 2017, pp. 662–676. Available at: http://doi.acm.org/10.1145/3140587.3062381, accessed 20.06.2018.

Интерактивный специализатор подмножества языка Java, основанный на методе частичных вычислений

[1] *И.А. Адамович <i.a.adamovich@gmail.com>*
[2] *Анд.В. Климов <klimov@keldysh.ru>*
[1] *Институт программных систем им. А.К. Айламазяна РАН,*
152021, Россия, Ярославская обл., с. Веськово, ул. Петра Первого, д. 4а
[2] *Институт прикладной математики им. М.В. Келдыша РАН,*
125047, Россия, Москва, Миусская пл., д. 4

Аннотация. Специализация — это оптимизация программ на основе использования наперёд заданной информации о значении части переменных. Методы специализации программ развиваются с 1970-х годов (смешанные вычисления, частичные вычисления, суперкомпиляция). Однако удивительно, что после трёх десятилетий разработанные специализаторы до сих пор не достигли того уровня, когда они станут пригодны для широкого практического применения. Возникает вопрос: в чём же причина? Наша гипотеза состоит в том, что задача специализации требуют гораздо большего участия человека в управлении процессом специализации, анализе результатов, проведении компьютерных экспериментов, чем в случае обычной оптимизации программы в компиляторах. Требуется погружение специализаторов в привычные для программистов интегрированные среды разработки, включая создание соответствующих диалоговых средств. В данной статье описываются результаты разработки и реализации методов интерактивной специализации на основе частичных вычислений для подмножества языка Java. Реализация выполнена в рамках популярной среды разработки (IDE) Eclipse. Разрабатываются сценарии человеко-машинного диалога с подсистемой специализации, интерактивные средства для составления задания на специализацию и управление процессом специализации. Приводится пример успешного применения разработанного специализатора. Остаточная программа работает в несколько раз быстрее чем исходная.

Ключевые слова: анализ программ; преобразование программ; интерактивная специализация программ; частичные вычисления; объектно-ориентированный язык; среда разработки программ

DOI: 10.15514/ISPRAS-2018-30(4)-2

Для цитирования: Адамович И.А., Климов Анд.В. Интерактивный специализатор подмножества языка Java основанный на методе частичных вычислений. Труды ИСП РАН, том 30, вып. 4, 2018 г., стр. 29-44 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(4)-2

Список литературы

[1]. Jones N.D., Sestoft P. and Søndergaard H. An experiment in partial evaluation: the generation of a compiler generator. Rewriting Techniques and Applications, Lecture Notes in Computer Science, J.-P. Jouannaud, (Ed.), vol. 202. Springer-Verlag, 1985, pp. 124–140

[2]. Jones N.D., Gomard C.K., and Sestoft P. Partial Evaluation and Automatic Program Generation. Prentice-Hall, 1993. Доступно по ссылке: http://www.itu.dk/~sestoft/pebook/pebook.html, дата обращения: 20.06.2018

[3]. Futamura Y. Partial evaluation of computation process — an approach to a compiler-compiler. Systems, Computers, Controls, vol. 2, no. 5, 1971, pp. 45–50

[4]. Futamura Y. Partial evaluation of computation process — an approach to a compiler-compiler. Higher-Order and Symbolic Computation, vol. 12, no. 4, Dec 1999, pp. 381–391. Updated and revised version of [3]. Доступно по ссылке: http://doi.org/10.1023/A:1010095604496, дата обращения: 20.06.2018

[5]. Futamura Y. EL1 Partial Evaluator (Progress Report). Center for Research in Computing Technology, Harvard University, Tech. Rep., 1973. Доступно по ссылке: http://fi.ftmr.info/PE-Museum/EL1.PDF, дата обращения: 20.06.2018

[6]. Turchin V.F. The concept of a supercompiler. ACM Transactions on Programming Languages and Systems, vol. 8, no. 3, 1986, pp. 292–325

[7]. Turchin V.F. Supercompilation: techniques and results. Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, June 25-28, 1996. Proceedings, Lecture Notes in Computer Science, D. Bjørner, M. Broy, and I.V. Pottosin, (Eds.), vol. 1181. Springer, 1996, pp. 227–248

[8]. Eclipse Foundation. Eclipse Integrated Development Environment (IDE). Доступно по ссылке: https://www.eclipse.org, дата обращения: 20.06.2018

[9]. Eclipse Foundation. Eclipse Java development tools (JDT). Доступно по ссылке: https://www.eclipse.org/jdt, дата обращения: 20.06.2018

[10]. Klimov Yu.A. An approach to polyvariant binding time analysis for a stack-based language. First International Workshop on Metacomputation in Russia, Proceedings. Pereslavl-Zalessky, Russia, July 2–5, 2008. Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008, pp. 78–84. Доступно по ссылке: http://meta2008.pereslavl.ru/accepted-papers/paper-info-6.html, дата обращения: 20.06.2018

[11]. Климов Ю.А. Особенности применения метода частичных вычислений к специализации программ на объектно-ориентированных языках. Препринты ИПМ им. М.В. Келдыша, № 12, 2008. Доступно по ссылке: http://library.keldysh.ru/preprint.asp?id=2008-12, дата обращения: 20.06.2018

[12]. Климов Ю.А. Возможности специализатора CILPE и примеры его применения к программам на объектно-ориентированных языках. Препринты ИПМ им. М.В. Келдыша, № 30, 2008. Доступно по ссылке: http://library.keldysh.ru/preprint.asp?id=2008-30, дата обращения: 20.06.2018

[13]. Климов Ю.А. SOOL: объектно-ориентированный стековый язык для формального описания и реализации методов специализации программ. Препринты ИПМ им. М.В. Келдыша, № 44, 2008. Доступно по ссылке: http://library.keldysh.ru/preprint.asp?id=2008-44, дата обращения: 20.06.2018

[14]. Климов Ю.А. Специализатор CILPE: анализ времен связывания. Препринты ИПМ им. М.В. Келдыша, № 7, 2009. Доступно по ссылке: http://library.keldysh.ru/preprint.asp?id=2009-07, дата обращения: 20.06.2018

[15]. Климов Ю.А. Специализатор CILPE: генерация остаточной программы. Препринты ИПМ им. М.В. Келдыша, № 8, 2009. Доступно по ссылке: http://library.keldysh.ru/preprint.asp?id=2009-08, дата обращения: 20.06.2018

[16]. Климов Ю.А. Специализатор CILPE: доказательство корректности. Препринты ИПМ им. М.В. Келдыша, № 33, 2009. Доступно по ссылке: http://library.keldysh.ru/preprint.asp?id=2009-33, дата обращения: 20.06.2018

[17]. Климов Ю.А. Специализация программ на объектно-ориентированных языках методом частичных вычислений. Дис. к.ф.-м.н., Институт прикладной математики им. М.В. Келдыша РАН, Москва, Россия, ноябрь 2009, 183 стр. Доступно по ссылке: http://pat.keldysh.ru/~yura/publications/2009.10-Klimov-Disser-Specializacia_programm_na_ob'ektno-orientirovannyx_yazykah.pdf, дата обращения: 20.06.2018

[18]. Климов Ю.А. Специализатор CILPE: частичные вычисления для объектно-ориентированных языков. Программные системы теория и приложения, № 3(3), 2010, стр. 13–36 Доступно по ссылке: http://psta.psiras.ru/read/psta2010_3_13-36.pdf, дата обращения: 20.06.2018

[19]. Bulyonkov M.A. and Kochetov D.V. Practical aspects of specialization of Algol-like programs. Dagstuhl Seminar on Partial Evaluation, Lecture Notes in Computer Science, O. Danvy, R. Gluck, and P. Thiemann, (Eds.), vol. 1110. Springer, 1996, pp. 17–32

[20]. Ershov A.P. and Itkin V.E. Correctness of mixed computation in Algol-like programs. MFCS, Lecture Notes in Computer Science, J. Gruska, (Ed.), vol. 53. Springer, 1977, pp. 59–77

[21]. Andersen L.O. Program analysis and specialization for the C programming language. Ph.D. dissertation, DIKU, University of Copenhagen, May 1994, (DIKU report 94/19)

[22]. Andersen L.O. Binding-time analysis and the taming of C pointers. Proceedings of the 1993 ACM SIGPLAN symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '93). ACM, 1993, pp. 47-58. Доступно по ссылке: http://dx.doi.org/10.1145/154630.154636, дата обращения: 20.06.2018

[23]. Consel C., Lawall J.L., and Meur A.-F.L. A tour of Tempo: a program specializer for the C language. Sci. Comput. Program., vol. 52, no. 1-3, 2004, pp. 341–370

[24]. Meur A.L., Lawall J.L. and Consel C. Towards bridging the gap between programming languages and partial evaluation. Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM ’02), Portland, Oregon, USA, January 14-15, 2002, P. Thiemann, (Ed.). ACM, 2002, pp. 9–18. Доступно по ссылке: http://doi.acm.org/10.1145/503032.503033, дата обращения: 20.06.2018

[25]. Schultz U.P., Lawall J.L. and Consel C. Automatic program specialization for Java. ACM Trans. Program. Lang. Syst., vol. 25, no. 4, 2003, pp. 452–499

[26]. Muller G., Moura B., Bellard F. and Consel C. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS), June 16-20, 1997, Portland, Oregon, USA, S. Vinoski, (Ed.). USENIX, 1997, pp. 1–20. Доступно по ссылке: http://www.usenix.org/publications/library/proceedings/coots97/muller.html, дата обращения: 20.06.2018

[27]. Shali A. and Cook W.R. Hybrid partial evaluation. SIGPLAN Not., vol. 46, no. 10, Oct. 2011, pp. 375–390. Доступно по ссылке: http://doi.acm.org/10.1145/2076021.2048098, дата обращения: 20.06.2018

[28]. Ji R. and Bubel R. PE-KeY: A partial evaluator for Java programs. Proceedings of the 9th International Conference on Integrated Formal Methods, IFM’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 283– 295. Доступно по ссылке: http://dx.doi.org/10.1007/978-3-642-30729-4_20, дата обращения: 20.06.2018

[29]. Ahrendt W., Beckert B., Bubel R., Hahnle R., Schmitt P.H. and Ulbrich M., (Eds.). Deductive Software Verification — The KeY Book — From Theory to Practice. Lecture Notes in Computer Science. Springer, 2016, vol. 10001. Доступно по ссылке: https://doi.org/10.1007/978-3-319-49812-6, дата обращения: 20.06.2018

[30]. Würthinger T., Wimmer C., Wöß A., Stadler L., Duboscq G., Humer C., Richards G., Simon D., and Wolczko M. One VM to rule them all. Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013. New York, NY, USA: ACM, 2013, pp. 187–204. Доступно по ссылке: http://doi.acm.org/10.1145/2509578.2509581, дата обращения: 20.06.2018

[31]. Würthinger T., Wimmer C., Humer C., Wöß A., Stadler L., Seaton C., Duboscq G., Simon D., and Grimmer M. Practical partial evaluation for high-performance dynamic language runtimes. SIGPLAN Not., vol. 52, no. 6, Jun. 2017, pp. 662–676. Доступно по ссылке: http://doi.acm.org/10.1145/3140587.3062381, дата обращения: 20.06.2018