Partial Evaluation for Software Engineering

C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi,
IRISA / INRIA - University of Rennes, France
and
J. Lawall
Oberlin College, Ohio
and
J. Noyé
École des Mines de Nantes, France

Up to now, partial evaluation has focused on the specialization process. Less attention has been devoted to validating the technology on concrete applications. This paper presents methods that are essential to integrate partial evaluation into the software engineering process, either explicitly by declaring specialization opportunities, or implicitly by using software architectures and mechanisms that are known to expose predictable specialization opportunities.

1. BEYOND THE SPECIALIZATION PROCESS

Partial evaluation has been intensively studied in the past twenty years. It has made major advances regarding the understanding of specialization from both a semantic and an implementation viewpoint. Many variations with respect to various language paradigms and features have been explored. Though prototype implementations have been putting theory into practice for several (mainly declarative) programming languages, they have only been applied to small programs. Less attention has been devoted to validating the technology on concrete applications.

Recently, however, attention has turned toward making a real proof of concept, so as to convince potential end-users in various communities. To do so, a partial evaluator has to successfully specialize large pieces of code, written in languages used by the software industry. Widely-used programming languages like Fortran and C are now being targeted for the development of partial evaluators [Baier et al. 1994; Andersen 1994; Consel et al. 1996].

However, as opposed to an optimizing compiler which calls for little parameterization, a specializer is a complex tool that requires specific expertise from the programmer to guide the optimization process. As a result, partial evaluation is seldom used outside of its own community.

This paper presents essential methods for partial evaluation to be actually integrated in a software engineering process, either explicitly by declaring specialization opportunities, or implicitly by using software architectures and mechanisms that are known to expose predictable specialization opportunities. Another use of partial evaluation for software engineering, not treated in this paper, is program understanding [Blazy and Facon 1997].

Name: Compose project (http://www.irisa.fr/compose)
Address: IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France

2. EXPLICIT SPECIALIZATION

As partial evaluators become more powerful, they also become more difficult to use. The user must identify the portion of code to specialize and describe the invariants to which specialization should be carried out. Then the user must choose between traditional compile-time specialization and new forms of specialization such as runtime specialization [Consel and Noël 1996], incremental specialization [Consel et al. 1993], and data specialization [Knoblock and Ruf 1996]. Finally, specialized routines must be installed, so that they are used in valid specialization contexts.

To make this process accessible to a non-expert, a simple interface must be provided, allowing a high-level description of what, when, and how to specialize. *Specialization classes* [Volanschi et al. 1997] are an example of such a specification that automatically exploits and manages specialization. Specialization classes are fully integrated in the object-oriented paradigm. They introduce declarations to the Java language that enable a programmer to specify how programs should be specialized for a particular usage pattern.

This approach has been implemented as a compiler from our extended language into standard Java. It is currently being used for experimentation with Java specialization through Harissa (a Java to C translator) [Muller et al. 1997] and Tempo (a C specializer) [Consel et al. 1996; Consel et al. 1998].

Specialization classes allow a programmer to declare specialization opportunities. Beyond this explicit approach, one may wonder whether opportunities can be exposed implicitly.

3. IMPLICIT SPECIALIZATION OF SOFTWARE ARCHITECTURES

Experience has shown that specialization works well on generic programs. This observation suggests that good specialization can be guaranteed for some programming styles and software architectures. Thus, partial evaluation could be embedded in a built-in mechanism responsible for the efficient instantiation of a software architecture.

A software architecture expresses how systems should be built from various components and how those components should interact. A key feature of software architectures is flexibility, resulting in re-usability, extensibility, adaptability. Because flexibility is present not only at the design level but also in the implementation, it may introduce a performance overhead. Sources of inefficiency can be identified in the integration of data (exchanged or shared) and control (means of communication) between components. Partial evaluation has been shown to improve the performance of a wide range of software architectures and mechanisms [Marlet et al. 1997]:

Selective Broadcast. In the selective broadcast (or *implicit invocation*) architecture [Shaw and Garlan 1996], components are independent agents that interact with each others by subscribing to certain types of events and sending broadcast messages. Specializing with respect to a subscription converts broadcasting into explicit calls to registered agents.

Pattern Matching. In an environment like Field [Reiss 1990], pattern matching is used to select broadcast events and decode message arguments. Specializing the pattern matcher and the decoder with respect to the pattern generates automata-like routines.

Software Layers. A layered system is a hierarchical organization of a program where each layer provides services to the layer above it and acts as a client to the layer below. An example is Sun's implementation of the Remote Procedure Call (RPC) protocol. If we specialize with respect to the data interface description, partial evaluation tightly merges the micro-layers that perform the encoding/decoding of data to/from a network independent representation, resulting in simple memory transfers [Muller et al. 1997].

Interpreters. Scripting languages glue together powerful components (building blocks) written in traditional systems programming languages. For flexibility and simplicity, they are often interpreted. If we specialize with respect to the script, the partial evaluation of the interpreter acts as a compiler [Jones 1996]. This approach applies to domain-specific languages as well, with the additional specialization of the components [Thibault and Consel 1997]. It has been successfully put into practice for the automatic generation of efficient video card drivers [Thibault et al. 1997].

Generic Libraries. Complex data structures consist of shape as well as actual content information. Routines in very generic libraries need to test the validity of such arguments and complete bounds checking before performing the actual service. Specializing with respect to the shape of a data structure eliminates verifications: the safety interface layer is compiled away.

The above invariants need not be available at compile time in order to allow successful optimization: partial evaluation can be performed at runtime as well [Consel and Noël 1996]. In contrast with unstructured programming, the improvement here can be predicted and guaranteed. Since partial evaluation is automatic, it does not defeat the goals of software engineering: performance is improved while flexibility is retained.

4. CONCLUSION

In order to bridge the gap between software engineers, programmers and partial evaluation, we have proposed two directions. On the one hand, explicit partial evaluation is specified using a high-level description that describes specialization opportunities and hides the intricacies of a partial evaluator. On the other hand, partial evaluation is implicitly guaranteed when using a wide range of software architectures.

Both tracks are actively pursued within the Compose project, as we foresee that partial evaluation will not be successful outside of its community until, paradoxically, it has completely disappeared from the scene.

REFERENCES

ANDERSEN, L. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph. D. thesis, Computer Science Department, University of Copenhagen. DIKU Technical Report 94/19.

ASE'97. 1997. *Conference on Automated Software Engineering* (Lake Tahoe, Nevada, Nov. 1997). IEEE Computer Society.

BAIER, R., GLÜCK, R., AND ZÖCHLING, R. 1994. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (Orlando, FL, USA, June 1994), pp. 119–132. Technical Report 94/9, University of Melbourne, Australia.

BLAZY, S. AND FACON, P. 1997. Application of formal methods to the development of a software maintenance tool. In *Conference on Automated Software Engineering* (Lake Tahoe, Nevada, Nov. 1997), pp. 162–171. IEEE Computer Society.

CONSEL, C., HORNOF, L., LAWALL, J., MARLET, R., MULLER, G., NOYÉ, J., THIBAULT, S., AND VOLANSCHI, N. 1998. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*. To appear.

CONSEL, C., HORNOF, L., NOËL, F., NOYÉ, J., AND VOLANSCHI, E. 1996. A uniform approach for compile-time and run-time specialization. In O. DANVY, R. GLÜCK, AND P. THIEMANN Eds., *Partial Evaluation, International Seminar, Dagstuhl Castle*, Number 1110 in Lecture Notes in Computer Science (Feb. 1996), pp. 54–72.

CONSEL, C. AND NOËL, F. 1996. A general approach for run-time specialization and its application to C. In *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages* (St. Petersburg Beach, FL, USA, Jan. 1996), pp. 145–156. ACM Press.

CONSEL, C., PU, C., AND WALPOLE, J. 1993. Incremental specialization: The key to high performance, modularity and portability in operating systems. In *Partial Evaluation and Semantics-Based Program Manipulation* (Copenhagen, Denmark, June 1993), pp. 44–46. ACM Press. Invited paper.

DANVY, O., GLÜCK, R., AND THIEMANN, P. Eds. 1996. *Partial Evaluation, International Seminar, Dagstuhl Castle*, Number 1110 in Lecture Notes in Computer Science (Feb. 1996).

JONES, N. 1996. What *not* to do when writing an interpreter for specialisation. In O. DANVY, R. GLÜCK, AND P. THIEMANN Eds., *Partial Evaluation, International Seminar, Dagstuhl Castle*, Number 1110 in Lecture Notes in Computer Science (Feb. 1996), pp. 216–237.

KNOBLOCK, T. AND RUF, E. 1996. Data specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation* (May 1996), pp. 215–225. ACM SIGPLAN Notices, 31(5). Also TR MSR-TR-96-04, Microsoft Research, February 1996.

MARLET, R., THIBAULT, S., AND CONSEL, C. 1997. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering* (Lake Tahoe, Nevada, Nov. 1997), pp. 183–192. IEEE Computer Society.

MULLER, G., MOURA, B., BELLARD, F., AND CONSEL, C. 1997. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems* (Portland (Oregon), USA, June 1997), pp. 1–20. Usenix.

MULLER, G., VOLANSCHI, E., AND MARLET, R. 1997. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Amsterdam, The Netherlands, June 1997), pp. 116–125. ACM Press.

REISS, S. P. 1990. Connecting tools using message passing in the filed environment. *IEEE Software 7*, 4 (July), 57–66.

SHAW, M. AND GARLAN, D. 1996. *Software Architecture*. Prentice Hall.

THIBAULT, S. AND CONSEL, C. 1997. A framework of application generator design. In *Proceedings of the Symposium on Software Reusability* (May 1997), pp. 131–135.

THIBAULT, S., MARLET, R., AND CONSEL, C. 1997. A domain-specific language for video device drivers: from design to implementation. In *Conference on Domain Specific Languages* (Santa Barbara, CA, Oct. 1997), pp. 11–26. Usenix.

VOLANSCHI, E., CONSEL, C., MULLER, G., AND COWAN, C. 1997. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings* (Atlanta, USA, Oct. 1997), pp. 286–300. ACM Press.