Higher-Order and Symbolic Computation 12, 377–380 (1999)

Partial Evaluation of Computation Process, Revisited

YOSHIHIKO FUTAMURA futamura@futamura.info.waseda.ac.jp
Department of Information and Computer Science, Waseda University, 3-4-1, Ohkubo, Shinjuku, Tokyo, Japan 169-8555

The original paper [3] of this reproduction is a translation of a Japanese paper published in Vol. 54-C, No. 8 of *Transactions of the Institute of Electronics and Communication Engineers of Japan*, pp. 721–728, in August 1971. In this reprint, I have corrected a few typos, and refined some sentences to make them more readable. Here is a FAQ list about partial evaluation and myself.

Q1: How did the idea of self-application come to you?

A1: After graduating from the Department of Mathematics at Hokkaido University (in Sapporo) in 1965, I engaged in developing an interpreter and a compiler for Lisp 1.5 at Hitachi Central Research Laboratory. The work was done based only on McCarthy's manual [12]. When I was implementing the compiler, I realized that I was just looking at the interpreter written in the manual. Then the idea that a compiler could be generated from an interpreter came to me. I think this idea is fundamental. Other results such as partial evaluation and self-application follow the idea with good luck.

Q2: Did you really implement a self-applicable partial evaluator in 1971 and how efficient was it?

A2: I implemented a partial evaluator α_1 in about 100 lines of Lisp 1.5. By design, its implementation language was a subset of its input language, and therefore α_1 was self-applicable in principle. In practice, however, self-applying α_1 was somewhat unsatisfying and even problematic. For example, and as witnessed by the residual code at the end of Section 5, α_1 did not pass Neil Jones's optimality test [9], which says that specializing an interpreter should remove its interpretive overhead completely. But of course, that optimality test was developed much later.

At the time, I was mainly struggling with issues such as ensuring termination and treating static side effects correctly—issues that to the best of my knowledge are still generally unsolved today, independently of self-application. In that sense, α_1 was overly ambitious. Also, α_1 was an online partial evaluator, which in hindsight did not help for self-application, though of course nobody realized that at the time. And indeed it took nearly 15 years and a new partial-evaluation technology (the so-called "offline" partial evaluation) before an efficient self-applicable partial evaluator could

see the light of day, in the Mix project at DIKU [10, 11]. At that occasion, Harald Søndergaard compared triple self-application to driving a truck with two trailers in reverse. I certainly concur.

Q3: Where and when did the term "Futamura projection" become in use?

A3: In 1980, in Vol. 12, No. 14 of the Japanese journal *Bit*, Andrei Ershov wrote an article entitled "Futamura Projections." However, I believe, the term was first used in English at PEMC'87, and documented in its proceedings [13]. My paper (i.e., this reprint) does not deal with the third Futamura Projection which shows that $\alpha(\alpha, \alpha)$ is a compiler-compiler. Actually, I did not realize the importance of this fact when I wrote "An Approach to a Compiler-Compiler" and thus I only reported the third projection in 1972 and 1973, while visiting Harvard University [4, 5]. (The introduction to [5] is reproduced in appendix.)

Q4: What is the relationship between partial evaluation and program transformation?

A4: Partial evaluation is a kind of program transformation. The partial evaluator α_1 (Section 4 (2) and (5)) implements what Burstall and Darlington have later called "folding" and "unfolding" [1].

Q5: Is a generating extension more efficient than a partial evaluator?

A5: Given a partial evaluator α and a program f, $\alpha(\alpha, f)$ yields the generating extension of f [13]. The term "generating extension" is due to Ershov [2]. As a specialized partial evaluator, a generating extension is usually more efficient than a general-purpose partial evaluator.

Q6: Is there a Fourth Futamura Projection?

A6: If you substitute α for *int* in the third projection, you obtain $\alpha(\alpha, \alpha)(\alpha) = \alpha(\alpha, \alpha)$ [6]. You may call this the fourth projection. At DIKU, where α is called "mix" after Ershov, I hear that the Fourth projection is nicknamed "the mixpoint" (the nickname is actually due to Peter Sestoft). The existence of a Fifth projection is hardly imaginable since there is nothing left to specialize with respect to.

Q7: How did you generalize partial evaluation?

A7: The partial evaluator described in this reprint uses only information given as actual parameters, i.e., input data. We proposed a method, called GPC (Generalized Partial Computation), that also exploits the logical structure of a source program. This makes it possible to determine some a priori unknown variables in source programs and to resolve conditionals, automatically [7, 8]. At Waseda University, the implementation of GPC is pursued under the support of the Research for the Future Program of the Japan Society for the Promotion of Science (JSPS).

Q8: Besides the Futamura projections, what else did you contribute to computer science?

A8: I have developed a Problem Analysis Diagram (PAD) which is an integrated method for program development. Around a diagrammatic method for structuring programs, PAD offers systematic design, coding and testing procedures. It has been included in an international standard in 1986 (ISO8631) and in a Chinese national standard in 1990 (GB13502).

Q9: What are you working on nowadays?

A9: Beside partial evaluation and program transformation, I am now working on programming methodology and on the generation of test data.

Acknowledgments

I am very grateful for the encouragement and stimulation received from many researchers including the late Andrei Ershov, K. Fuchi, Tom Cheatham, Valentin Turchin, Neil Jones, Dines Bjørner and Erik Sandewall. Thanks are also due to the editors of this journal for their enormous support for reproducing my old paper and writing this introduction.

Notes: Extract from the introduction of "EL1 Partial Evaluator (Progress Report)", January 23, 1973

The aim of this report is to provide information concerning work done, problems encountered, and difficulties overcome while the project for the EL1 partial evaluator has been underway since October, 1972.

The goal of the project is to implement a partial evaluator α for EL1 which has the properties described below. α is a partial evaluator such that

$$\alpha(int, s)(r) = int(s, r) \tag{1}$$

for an arbitrary program *int* and its arguments s and r. We can derive the following equations from 1.

$$\alpha(int, s)(r) = \alpha(\alpha, int)(s)(r) \tag{2}$$

$$= \alpha(\alpha, \alpha)(int)(s)(r) \tag{3}$$

If we consider *int*, s, and r as an interpreter, a source program and runtime data respectively, then $\alpha(int, s)$, $\alpha(\alpha, int)$, and $\alpha(\alpha, \alpha)$ can be considered as an object program, a compiler, and a compiler generator, respectively. In order to be useful, α has to have the following two properties.

1. α should evaluate as many portions of programs (e.g., *int* and α) as possible in order to save time during the final evaluation.
2. α should avoid working on portions of the program not needed in the final evaluation in order to save partial-evaluation time.

References

1. Burstall, R. and Darlington, J. A transformation system for developing recursive programs. *JACM* **24**(1) (1977) 44–67.

2. Ershov, A. On the essence of compilation. In *Formal Description of Programming Concepts*, Neuhold, E. (Ed.). North-Holland, 1978, 391–420.
3. Futamura, Y. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems Computers Controls* **2**(5) (1971) 45–50.
4. Futamura, Y. EL1 Partial Evaluator. Term paper manuscript, AM260, DEAP, Harvard University, 1972.
5. Futamura, Y. EL1 Partial Evaluator (Progress Report). Center for Research in Computing Technology, Harvard University, January 1973.
6. Futamura, Y. Partial computation of programs. In *RIMS Symposia on Software Science and Engineering*, E. Goto et al. (Ed.). LNCS 147, Springer, 1982.
7. Futamura, Y. and Nogi, K. Program Transformation Based on Generalized Partial Computation. US-Patent 5241678, August 31, 1993.
8. Futamura, Y., Nogi, K., and Takano, A. Essence of generalized partial computation. *Theoretical Computer Science* **90** (1991) 61–79.
9. Jones, N.D., Gomard, C.K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation*, Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.
10. Jones, N.D., Sestoft, P., and Søndergaard, H. An experiment in partial evaluation: the generation of a compiler generator. In *Rewriting Techniques and Applications*, Jouannaud, J.-P. (Ed.). Lecture Notes in Computer Science, Vol. 202, Springer-Verlag, 1985, pp. 124–140.
11. Jones, N.D., Sestoft, P., and Søndergaard, H. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation* **2**(1) (1989) 9–50.
12. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.
13. Morgensen, T. and Holst, K. Terminology. *New Generation Computing* **6** (1988) 303–307.