
Higher-Order and Symbolic Computation 12, 381–391 (1999)

Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler

YOSHIHIKO FUTAMURA
Central Research Laboratory, Hitachi, Ltd., Kokubunji, Tokyo, Japan 185

Abstract. This paper reports the relationship between formal description of semantics (i.e., interpreter) of a programming language and an actual compiler. The paper also describes a method to automatically generate an actual compiler from a formal description which is, in some sense, the partial evaluation of a computation process. The compiler-compiler inspired by this method differs from conventional ones in that the compiler-compiler based on our method can describe an evaluation procedure (interpreter) in defining the semantics of a programming language, while the conventional one describes a translation process.

Keywords: partial evaluation, program transformation, compiler, interpreter, Futamura projections

1. Introduction

It is known that there are two methods to formally describe the semantics (meaning) of programming languages. One of them is to describe the procedure by which the language to be defined is translated into another language whose semantics are already known, i.e., a description of a translator. The other is to describe a procedure evaluating the results of a statement belonging to the language to be defined (a source program), i.e., a description of an interpreter.

In a conventional compiler-compiler, the description of a translator is used to describe the semantics of a programming language. That is, the users of a conventional compiler-compiler have to write the translation program in terms of a translator description language in defining the semantics of a programming language.

The difficulty in writing a translator has been pointed out by Feldman [2] as follows:

"One of the most difficult concepts in translator writing is the distinction between actions done at translate time and those done at run time. Anyone who has mastered this difference has taken the basic step towards gaining an understanding of computer languages."

In describing the semantics of a programming language by an interpreter, it is not necessary to set up a distinction between those actions. Therefore, describing an interpreter seems easier than describing a translator. Actually, description by an interpreter is implicitly used at many places in the report on ALGOL 60 [6] and in manuals of many programming languages. The interpreters of such complex languages as ALGOL 60 and PL/I also have been described formally [3, 8].

*This is an updated and revised version of **Partial Evaluation of Computation Process—an Approach to a Compiler-Compiler by Yoshihiko Futamura**, originally published in "Systems.Computers.Controls", Volume 2, Number 5, 1971, pages 45–50. Reprinted by permission of John Wiley & Sons, Inc.

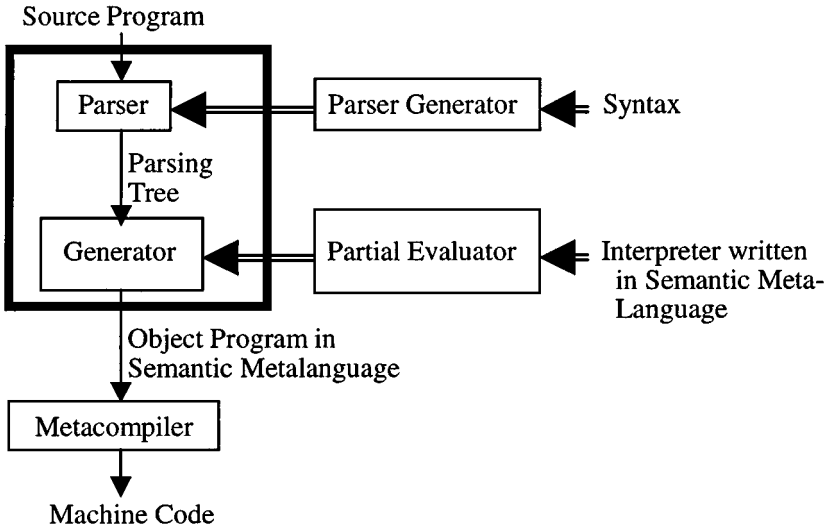

Figure 1. Structure of a compiler-compiler. The large bold-line block is the generated compiler. The object language of this compiler is a semantic metalanguage in which an interpreter is described. An object program is translated into machine codes by the metacompiler.

However, for reasons described in Section 3, a so-called interpreter is often not as efficient as a so-called compiler in language processing.

This paper describes an algorithm to automatically transform an interpreter to a compiler and its application to a compiler-compiler. The algorithm is a sort of partial evaluation procedure (see figure 1).

Partial evaluation of a computation process is by no means a new concept [4]. Even in programming languages, POP-2 [1] implies a somewhat similar concept called "partial application." Nevertheless, it is the author's belief that this paper is the first instance in which the concept is applied in a compiler-compiler. What kind of partial evaluation algorithm is applicable to a compiler-compiler? It is the purpose of this paper to probe the properties of that algorithm.

2. Partial evaluation

The following transformation is called a partial evaluation of a computation process with respect to variables $c_1, \ldots, c_m$, at the values $c'_1, \ldots, c'_m$. "In a computation process containing $m + n$ variables $c_1, \ldots, c_m, r_1, \ldots, r_n$, evaluate the portions of which can be evaluated using only the values $c'_1, \ldots, c'_m$ assigned to variables $c_1, \ldots, c_m$, respectively, and constants contained in . The portions which cannot be evaluated unless the values of the remaining variables are given are left intact. Thus, is transformed into a computation process having n variables. When the computation process thus obtained is evaluated for values $r'_1, \ldots, r'_n$ assigned to variables $r_1, \ldots, r_n$, respectively, its result is equivalent to the result of the evaluation of it for the values $c'_1, \ldots, c'_m$, $r'_1, \ldots, r'_n$ given to variables

$c_1, \ldots, c_m, r_1, \ldots, r_n$, respectively." We denote this transformation by the equation

$$(c'_1, \ldots, c'_m, r'_1, \ldots, r'_n) = \quad (\ , c'_1, \ldots, c'_m)(r'_1, \ldots, r'_n) \tag{1}$$

We call the "partial evaluation algorithm," $c_1, \ldots, c_m$ the "partial evaluation variables," and $r_1, \ldots, r_n$ the "remaining variables," respectively. We may refer to the usual evaluation as a total evaluation as opposed to a partial evaluation.

For example, consider the evaluation of a computation process given by the function $f(x, y) = x \times (x \times x + x + y + 1) + y \times y$ with the values $x = 1, y = 1, 2, \ldots, l$.

When we evaluate $f(1, y)$ for each value of y, i.e., when we execute

$x := 1$;
for $y := 1$ step 1 until l do $f[x, y] := x \times (x \times x + x + y + 1) + y \times y$

$3l$ multiplications and $4l$ additions are performed. Representing the times elapsed in addition and multiplication by a and m respectively, the above computation requires about $(4a + 3m)l$.

If we have $(f, 1)(y) = 1 \times (3 + y) + y \times y$ by partial evaluation of $f(x, y)$ with respect to $x = 1$, the elapsed time of the partial evaluation, e.g., k, is more than $2a + m$, i.e., $k > 2a + m$ (because the partial evaluation involves the evaluation of $x \times x + x + 1$).

If we execute

for $y := 1$ step 1 until l do $f[1, y] := 1 \times (3 + y) + y \times y$;

a computation time of about $(2a + 2m)l$ is required. Therefore, when the relation

$$k + (2a + 2m)l < (4a + 3m)l \quad or \quad \frac{k}{2a + m} < l$$

holds, the partial evaluation gives a faster computation.

3. Generation of a compiler from an interpreter

An interpreter of a programming language is a computation process containing variables. A sentence (source program) of the programming language is substituted for one of the variables as a value. Variables contained in an interpreter, e.g., int, are classified into two groups as follows. All variables to which a source program and information needed for syntax analysis and semantic analysis are given as values are classified as a group s. The other variables are classified as a group r. Here, int is assumed to have two variables s and r. The result of the partial evaluation of the interpreter with respect to s at a given value s' is $(\text{int}, s')(r)$. With r' assigned to r as a value, the following relation is derived from Eq. (1):

$$\text{int}(s', r') = \quad (\text{int}, s')(r') \tag{2}$$

If all the computations concerning s' have been performed at partial evaluation time, the generated computation process (int, s') does not contain the computation process for syntax and semantic analysis of the source program s'. Moreover, it brings about the same result as int(s', r') when it is evaluated for the data r'. Therefore, (int, s') can be viewed as a computation process which is translated from s' into the semantic metalanguage describing the interpreter. Namely, it can be regarded as an object program corresponding to s'.

If is partially evaluated with respect to int on the right side of Eq. (2), the following relation is derived:

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

(, int) can be considered to be a compiler because it generates an object program from s' operating on it.

Suppose has the following two properties.

p1. In partially evaluating a computation process , evaluates as many portions of as possible which can be evaluated only with constants and values given to partial evaluation variables.

p2. evaluates as few portions of as possible which are actually not evaluated when a generated computation process is evaluated with the values of remaining variables.

Property p1 reduces the computation time of the process generated by a partial evaluation when it is evaluated with the given value of remaining variables. Property p2 reduces the computation time of a partial evaluation.

If a partial evaluation algorithm somehow possesses both properties p1 and p2, it is more efficient to execute a source program once compiled than to interpret it directly when the source program contains such iterations as loops and recursive calls or is iteratively executed for many input data.

The simplest partial evaluation algorithm is the one which neglects property p1, i.e., the one which only substitutes given values for partial evaluation variables.

The algorithm $_1$ considering the property p1 and fitted for the partial evaluation of an interpreter is described in the rest of this section.

For ease of explanation, a computation process is represented by a graph such as that in figure 2. In figure 2, nodes ($\circ$) represent conditional branching points, branches (arrows) represent subcomputation processes not containing a branching point and a flow of control, and the leaves ($\bullet$) represent the termination points of the computation process. All nodes and branches are marked n_i and b_j (a different one is subscripted by a different number), respectively. Let b_1 denote the entry branch (there may be more than one entry branch, but we assume that only one is selected at partial evaluation time) and let m denote the total number of branches.

$_1$ determines partial evaluation variables and the remaining variables at each stage of the partial evaluation depending on the following two criteria:

(i) Partial evaluation variables are (1) partial evaluation variables of the preceding stage or (2) variables (or formal parameters of functions) to which values depending only on constants or partial evaluation variables of the preceding stage are assigned.

(ii) Variables other than partial evaluation variables are remaining variables.

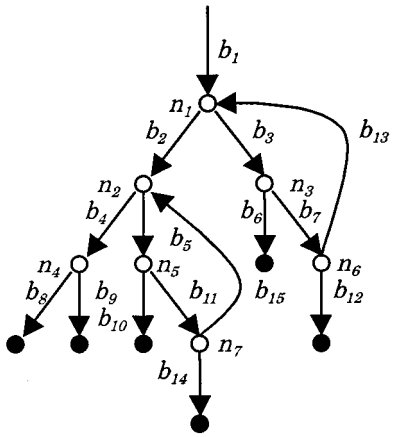

Figure 2. Graph representation of computation process (Here, $n_1, \ldots, n_7$ denote nodes and $b_1, \ldots, b_{15}$ denote branches).

The algorithm $_1$ is given by the five operations below (in the description of the algorithm, integer variables g, $j(1), \ldots, j(m)$ and a list variable L are used).

(1) Set each of g, $j(1), \ldots, j(m)$ to 1 and set L to nil. Proceed to operation (2).

(2) Allocate the first address of the space in which the result of partial evaluation of b_g is stored and memorize that address. (When the result is stored in the memory of a computer as a program, the first address is that of the program. When the result is written as a graph, the first address is that of a label attached to an entry point to branch b_g.) Namely, $_1$ enters the triplet $(b_g, S_g^{j(g)}, a_g^{j(g)})$ in list L, where $S_g^{j(g)}$ denotes the set of pairs of partial evaluation variables and their values (at the entry point of the $j(g)$th entry to b_g), and $a_g^{j(g)}$ denotes the first address of the space in which the result of the $j(g)$th partial evaluation of b_g is generated. Proceed to operation (3).

(3) Evaluate the portions of b_g which can be evaluated only with partial evaluation variables and constants. To those portions, attach marks indicating that they have already been evaluated. Let $b_g^{j(g)}$ denote the new computation process generated from b_g by this operation (Note that the first address of $b_g^{j(g)}$ is $a_g^{j(g)}$, and b_g is left intact). Increment the value of $j(g)$ by 1 and proceed to operation (4).

(4) If the process next to b_g (i.e., the arrowhead of b_g) is a termination symbol (•), stop the partial evaluation. If the process next to b_g is a conditional branching point $n_{k(i)}$, proceed to 4(A) or 4(B).

(A) If $n_{k(i)}$ can be evaluated only with the values of partial evaluation variables and constants, then select one of two branches based upon the value of $n_{k(i)}$. Let b_p express the branch selected. Set the value of g to p and proceed to operation (5).

(B) If $n_{k(i)}$ cannot be evaluated unless the values of remaining variables are given, then $n_{k(i)}$ is left intact. Let b_p and b_q denote two branches following $n_{k(i)}$. Set the value of g to p and proceed to operation (5). Next, set the value of g to q and proceed to operation (5).

(5) Examine list L to see whether there is a triplet whose first and second terms coincide with b_g and $S_g^{j(g)}$ respectively.

(A) If there is such a triplet, transfer control of the generated computation process to the position indicated by the third term a_g^x of the triplet (if a generated computation process is written on paper, draw an arrow to the place labeled a_g^x). Stop the partial evaluation.

(B) If there is no such triplet, return to operation (2).

Example 1. Suppose that the conditional branching points n_1, n_3 and n_6 can be evaluated only with partial evaluation variables and constants, and that each evaluation of n_1, n_3 and n_6 selects the branch b_3, b_7 and b_{12} respectively. Then, is transformed by $_1$ as described in figure 3.

Example 2. Consider the case in which n_1 and n_6 can be evaluated only with partial evaluation variables and constants, and the value of n_3 depends on the values of remaining variables. Let n_1 always select branch b_3 and let n_6 select branch b_{13} for the first time and select branch b_{12} for the second time. Then, is transformed by $_1$ as described in figure 4.

Example 3. In Example 2, if n_6 invariably selects b_{13}, $_1$ does not always terminate its computation and may generate such an infinite graph as described in figure 5. However, if n_6 always selects b_{13} simply because the partial evaluation variables of b_7 cyclically take the same values, the computation of $_1$ is terminated by operation (5). It produces such a result as described in figure 6 in the case when the values of partial evaluation variables of b_7 do not change. In partially evaluating an interpreter with respect to a source program which contains loops or recursive calls, the above case occurs. Therefore, operations (2) and (5) are essentially important for the compiler-compiler method described in this paper.

Example 4. Let us assume that n_1 in figure 7 depends on the remaining variables. In this case, if the repetitive partial evaluation of process b_3 does not produce the same S_3^x more than once, then an infinite graph will be generated. But in totally evaluating the process it is possible that, after b_3 has been computed several times, n_1 selects b_2 and the computation will terminate. If b_3 does not contain remaining variables but contains an infinite loop and if n_1 always selects b_2 in total evaluation, then it is a trivial example of a

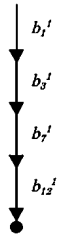

Figure 3. Example 1.

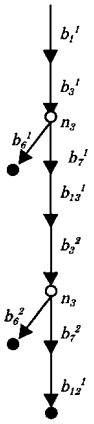

Figure 4. Example 2.

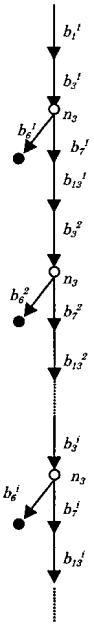

Figure 5. Example 3 (nonterminating case).

computation process whose total evaluation terminates but whose partial evaluation does not terminate.

The evaluation of those portions of a computation process which are not evaluated at total evaluation time, as in the last example, can be avoided by the following procedure. The

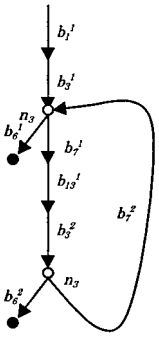

Figure 6. Example 3 (terminating case).

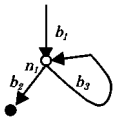

Figure 7. Example 4.

portions of a computation process (with the exception of conditional branching points) for which it is not known whether they are evaluated at total evaluation time (i.e., the portions following conditional branching points whose values depend on the values of remaining variables) are not evaluated at partial evaluation time, but the values are only substituted for the remaining variables. This procedure is necessary not only for the avoidance of wasteful evaluations at partial evaluation time but also to guard against the printing of erroneous statements and other troublesome portions of the interpreter which do not have to be evaluated at partial evaluation time (e.g., input-output operations).

We make an exception of conditional branching points in the foregoing procedure. In order to reduce the number of nodes and branches contained in the resulting computation process of a partial evaluation, we evaluate as many conditional branching points at partial evaluation time as possible. If the portions of a computation process, that follow conditional branching points containing remaining variables, also contain remaining variables, $_1$ is recursively applied to those portions. This is based on the idea that because the portions of a computation process containing remaining variables often include recursive calls to an interpreter, it is worthwhile to risk the partial evaluation of those portions. Therefore, functions, procedures and pseudo-variables which do not have to be evaluated at partial evaluation time must be marked and must be handled exceptionally

However, if we describe an interpreter carefully, we can avoid such a meaningless loop as the one described in Example 4. Therefore, the desired algorithm can be obtained by modifying $_1$ so that it evaluates all the portions of a computation process except those marked as unnecessary to be evaluated at partial evaluation time.

The partial evaluation algorithm has been described in the preceding discussion, but the details thereof have been omitted since they are quite different in each programming language describing a computation process.

Example 5. Partial evaluation of the LISP [5] function append[x;y] defined as

$$\text{append}[x; y] = [\text{null}[x] \rightarrow y; T \rightarrow \text{cons}[\text{car}[x]; \text{append}[\text{cdr}[x]; y]]]$$

Then,

$$_1[\text{append}; (A, B)][y] = \text{cons}[A; \text{cons}[B; y]]$$
$$_1[\text{append}; (A, B)][x] = \text{f}[x] = [\text{null}[x] \rightarrow (A, B);$$
$$T \rightarrow \text{cons}[\text{car}[x]; \text{f}[\text{cdr}[x]]]]$$

Note that the function name f in the above equation has been generated by the process (2) of $_1$.

Example 6. Partial evaluation of ALGOL program. Let a and b denote lists of integers (i.e., integer arrays). $a[0]$ and $b[0]$ contain the length of each list respectively. $a[1], a[2], \ldots, a[a[0]]$ and $b[1], b[2], \ldots, b[b[0]]$ contain the elements of the lists. The program [7] concatenating lists a and b is described below (wherein bigm denotes the subscript bound of array a).

```
begin if a[0] + b[0] > bigm then goto overfl;
      for k := 1 step 1 until b[0] do
      a[k + a[0]] := b[k]; a[0] := a[0] + b[0];
end
```

The result of partial evaluation of the above program with respect to b at $b[0] = 2, b[1] = 10, b[2] = 20$ is described below.

```
begin if a[0] + 2 > bigm then goto overfl;
      a[1 + a[0]] = 10;
      a[2 + a[0]] = 20;
      a[0] = a[0] + 2;
end
```

4. Discussion

(a) What is the criterion for the possibility of generating a compiler from an interpreter, i.e., a nontrivial sufficient termination condition of partial evaluation?
(b) Which parts of the object program (i.e. the result of partial evaluation of an interpreter with respect to a source program) are more efficient than the corresponding parts of the source program and to what extent? What are the characteristics of the object program and how may it be optimized (with respect to time and space)?
(c) Quantitatively, to what extent is describing an interpreter easier than describing a translator? Can we find a partial evaluation algorithm generating a compiler which is as efficient as a compiler generated from a translator?

(d) What kind of semantic metalanguage shall we use to describe an interpreter in order to achieve efficient partial evaluation of the interpreter?

At present, the author cannot answer the above questions clearly. It is considered that investigations along the following lines will solve those questions.

(1) Understanding structures of interpreters of programming languages.
(2) Development of a semantic metalanguage which can be efficiently compiled and by which we can easily describe the abstract syntax of programming languages, the states of abstract machines (stack, table, list, etc.) and their transitions, numerical computation, and list processing.
(3) Implementation of a complete partial evaluation algorithm for a specific semantic metalanguage.
(4) Theoretical study on the partial evaluation of computer programs.
(5) Optimization of semantic metalanguages.

The author has made a little progress on item (3). A partial evaluator which is almost equivalent to $_1$ has been implemented in LISP, and a compiler of program features [5] has been generated from the interpreter of program features by the partial evaluator. The compiler translates ALGOL-like programs written in the program features into an equivalent system of recursion equations. For example,

prog [[u; v];
  $u := n$;
$L1$ [null[u] $\rightarrow$ return[v]];
  $v :=$ cons[car[u]; v];
  $u :=$ cdr[u];
  go[$L1$]]

is translated into

g1[a] = g2[ppair[(U V); a]]
g2[a] = prog2[rplacd[assoc[U; a]; eval[N; a]]; g3[a]]
g3[a] = g4[a];
g4[a] = g5[a];
g5[a] = [eval[(NULL U); a] $\rightarrow$ eval[V; a]; $T \rightarrow$ g6[a]]
g6[a] = g7[a]
g7[a] = prog 2[rplacd[assoc[V; a]; eval[(CONS(CAR U) V); a]]; g8[a]]
g8[a] = prog 2 [rplacd[assoc[U; a]; eval[(CDR U); a]]; g9[a]]
g9[a] = g4[a]

where g1–g9 are the function names generated by the compiler and g1[a] is the object program. Superfluous equations such as g3[a] = g4[a], g4[a] = g5[a], etc., can be avoided by optimization of the semantic metalanguage (in this case, LISP).

5. Conclusion

The compiler generation method described in this paper is still in the conceptual stage. It remains to determine whether or not the method can be put to practical use in the near future. However, the author hopes that this paper explains the relationship between formal methods of programming language description and actual compilers. It is also hoped that this paper makes a contribution to the study of a compiler-compiler.

References

1. Burstall, R.M. and Popplestone, R.J. POP-2 reference manual. *Machine Intelligence* (2) (1968) 205–249.
2. Feldman, J.A. Formal Semantics for Computer-Oriented Languages. Technical Report, Comput. Ctr., Carnegie Institute of Technology, 1964.
3. Lauer, P. Formal Definition of ALGOL 60. Technical Report TR 25.088, IBM Laboratory Vienna, 1968.
4. Lombardi, L.A. *Advances in Computers*, Vol. 8. Academic Press, New York, 1967.
5. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I. *LISP 1.5 Programmer's Manual*, M.I.T. Press, 1962.
6. Naur, P. (Ed.). Revised report on the algorithmic language ALGOL 60. *Comm. of ACM* (6) (January 1963) 1–17.
7. Rutishauser, H. *Description of ALGOL 60*. Springer, 1967.
8. Walk, K., Alber, K., Bandat, K., Bekic, H., Chroust, Gerhard, Kudielka, V., Oliva, P. and Zeisel, G. Abstract syntax and interpretation of PL/I. Technical Report TR 25.082, IBM Laboratory Vienna, 1968.