Partial Evaluation of Computation Process and its Application to Compiler Generation

Yoshihiko Futamura,
Central Research Laboratory, Hitachi, Ltd.
Kokubunji, Tokyo, Japan.

ABSTRACT:

This paper discusses algorithms for evaluating portions of a computation process (program) on the basis of given partial information. Let us suppose a case where a computation process containing variables is being evaluated, where iterative computations are performed in which the values of the process variables are changed, and where there are numerous iterations. In such a case, if an algorithm of this type is used, it will be possible to transform the computation process so as to reduce the time complexity in the computation, making the computation more convenient than it would be if the process were to be carried out without any transformation at all. Furthermore, in cases when the semantics of a programming language is given by an interpreter, it will also be possible to generate a compiler to translate the program belonging to that language into a program in the language describing the interpreter.
KEY WORDS AND PHRASES: computation process, partial evaluation, algorithm, programming language, interpreter, compiler.

CR CATEGORIES: 4.12, 4.13, 5.23, 5.24

1 Introduction

This paper discusses algorithms for evaluating portions of a computation process[1] on the basis of given partial information. It is believed that this report is the first general consideration ever to be devoted to the partial evaluation of computation processes. Consider the case when a computation process containing variables,[2] in which the initial values of these variables are changed, is performed iteratively, or the case when the computation process which contains iterations such as loops or recursive calls is performed. When the number of iterations increases, it will then be possible to transform the computation process so as to reduce the time complexity in the computation, making the computation more convenient than it would be if the process were to be carried out without any transformations at all.

For instance, let us suppose that the following computation process is given:

1) By "computation processes" are meant Turing machines, recursive functions, arithmetic expressions, computer programs, graphs, etc.

2) By "variables" are meant the tape expressions and internal states of Turing machines; variables in recursive functions, graphs, and arithmetic expressions; the program parameters of computer programs; etc.

$$f(x, y) = x * (x * x + x + y + 1) + y * y$$

Let us presume that the values of x and y are given as

$x = 1, 2, \ldots, n$

$y = 1, 2, \ldots, l$

and that iterative computations are performed for them. In this case, one will evaluate[3] f(x, y) with respect to the values of x and y. In other words, the following is executed:

```
for x:=1 step 1 until n do   for y:=1 step 1
until ℓ do f[x, y]: = x * (x * x + x + y + 1) + y * y
```

In this case, altogether 3nl multiplications and 4nl additions will be performed. Consequently, let us express by m and a the time complexity required in the multiplication and addition, respectively. In this case, the total number of processes will be $(3m + 4a)nl$. With respect to x_0, the value of x, define $fx_0(y)$ as follows. That is, one evaluates, from the computation processes of f(x, y), those portions which can be evaluated with only the value of x(that is, x_0) and the constants. These portions are, $x * x + x + 1$. The portions which cannot be

3) Evaluation of a computation process means to execute the computations, giving the initial values of the variables.

evaluated unless the value of y is given are left untouched. That which is obtained in this way is treated as a new computation process. In other words, the following is obtained:

$$f_1(y) = 1 * (3 + y) + y * y$$
$$f_2(y) = 2 * (7 + y) + y * y$$

Now let us suppose that one has first sought fx(y) with respect to x = 1, 2, ..., n, and that one has next executed the following:

```
for x:=1  step 1 until n do    for y:=1  step 1 until 1
do f[x, y]: = fx(y)
```

Let us use k to express the time complexity needed to transform f(x, y) into fx(y). In this case, $k \geq m + 2a$. In the new computation processes, the number of processes required here for all of the f(x, y) computations will be:

$$(2m + 2a)nl + nk$$

Consequently, in cases when

$$l > k/(m + 2a)$$

rather than continuing the iterative calculations of $f(x, y)$ without modification, it will be preferable to compute after transforming into $fx(y)$. If this is done, it will be possible to reduce the time complexity of processes.

In this manner, when x_0, the value of variable x, is given, one evaluates the portion of $f(x, y)$ which can be evaluated by using x_0 and the constants contained in the computation process $f(x, y)$. Thus, the transformations for obtaining computation process $fx_0(y)$ are called the partial evaluation at value x_0 with respect to variable x of $f(x, y)$. In this case, the partial evaluation is a sort of algebraic manipulation.

Generally speaking, transformations of the following type are termed partial evaluation of the computation process at value $c'_1, \ldots, c'_m$ with respect to variable $c_1, \ldots, c_m$. "In a computation process containing $m + n$ variables $c_1, \ldots, c_m, r_1, \ldots, r_m$, the values $c'_1, \ldots, c'_m$ are assigned to variables $c_1, \ldots, c_m$, and evaluation is performed of the portions which can be evaluated with only these values and the constants included in the computation process. The portions which cannot be evaluated unless the values of the remaining variables are given are left untouched. In this manner, the original computation process is transformed into a computation process having n variables. The computation process obtained in this manner is evaluated by assigning values $r'_1, \ldots, r'_n$ to the variables $r_1, \ldots, r_n$ contained therein. When this is done, the evaluation will coincide with the evaluation obtained when the values $c'_1, \ldots, c'_m$ and $r'_1, \ldots, r'_n$ were assigned to the variables in the original computation process." (In Chapter 2 of this paper a precise definition of partial evaluation is provided, and it is shown that it is not always possible to perform partial evaluation for any computation process and for any combination of variables.)

One of the most important applications of partial evaluation is compiler generation when the semantics of the programming language is given by an interpreter. The compiler will in this case translate the sentences belonging to the programming language (the program) into a program in the semantic metalanguage dictating the interpreter.

The interpreter of a programming language is a computation process containing variables. One of its variables corresponds to the sentence belonging to the programming language (the program), which is given as a value. The values of the other variables are given as the information necessary for evaluating this sentence. From among the variables contained in the interpreter, e.g. int, let us represent all of the variables corresponding to the sentence belonging to the language, or corresponding to the information necessary for syntax analysis or semantic analysis of the sentence, as c_1, ..., c_m (assuming that the sentence is given in c_1). Also let us represent the other variables as r_1, ..., r_n. Assigning values c'_1, ..., c'_m to c_1, ..., c_m, let us partially evaluate the interpreter at c'_1, ..., c'_m with respect to c_1, ..., c_m. Thus, let us suppose that we obtain in this way:

$$\text{int}_{c_1', \ldots, c'_m}(r_1, \ldots, r_n)$$

In view of the nature of partial evaluation, in $\text{int}_{c_1', .., c'_m}(r_1, \ldots, r_n)$, the evaluation has been completed only concerning c'_1, ..., c'_m and the constants. In other words, the syntax analysis and the semantic analysis of the sentence has been completed. Furthermore, the expression

$$\text{int}_{c_1', \ldots, c'_m}(r_1', \ldots, r'_n) = \text{int}(c_1', \ldots, c'_m, r_1', \ldots, r'_n)$$

applies to any r'_1, ..., r'_n. Consequently, in $\text{int}_{c_1', \ldots, c'_m}(r_1, \ldots, r_n)$, sentence c'_1 is thought to be translated into a program in the language dictating the interpreter (the semantic metalanguage). r'_1, ..., r'_n can be thought to be the values needed when the translated program is running. Consequently, it is thought that, by performing partial evaluation of the interpreter, one can compile the sentences belonging to the language into a program in the semantic metalanguage. Utilizing this characteristic feature of partial evaluation, one generates the comiler of the programming language from the interpreter.

In order to eliminate ambiguity in the discussion of the foregoing matters in this paper, the computation processes are expressed in terms of LISP functions [1] in the S-expression, and the variables are expressed as their bound variables (that is, λ -variables). As long as there is a well formulated formalism adequate for expressing the computation process, any of the following may be used: Turing machines, partial recursive functions, graphs, arithmetic expressions, or computer programs. In spite of this, the LISP formalism has been selected in this paper for the following reasons. First, the semantic meanings are simple and clear. Second, by using induction [2], the correctness of the algorithms can be proven formally. Third, computation processes dictated in LISP can be put into an electronic computer without modification.

In Chapter 2 of this paper, a precise definition of partial evaluation of computation processes is given, and its characteristic features are investigated.

In Chapter 3, the algorithms of partial evaluation are described informally. In Chapter 4, the method of generating the compiler from the interpreter of the programming language is described, and the relationships between the interpreter and the compiler are investigated. Although this method represents a new approach to compiler-compiler, this paper does not touch upon its application to one.

A formal description of the partial evaluation algorithms is given in Appendix 1. Formal proofs of their correctness can be made, for the most part, by means of the induction method given in Reference [2], although there still remain some portions which cannot be proven formally. Therefore this paper does not touch upon the proof. In Appendix 2 is given an example of compiler generation by the method described in Chapter 4.

2 Partial Evaluation of the Computation Process

The S-expression [1] of LISP is used to express the computation process. Its interpretation is given by the universal function apply of pure LISP [1].

This report defines that two m-expressions are weakly equivalent [3] when they, $p[x_1; \ldots\ldots; x_n]$ and $q[x_1; \ldots\ldots; x_n]$, satisfy the following condition, and represent it as

$$p[x_1; \ldots\ldots; x_n] \underset{w}{=} q[x_1; \ldots\ldots; x_n]$$

"Letting domain of p and q to be Dp and Dq, $p[y_1; \ldots\ldots; y_n] = q[y_1; \ldots\ldots; y_n]$ is held for all elements $[y_1; \ldots\ldots; y_n]$ of $Dp \cap Dq$".

where the equality sign stands for equality of S-expression. The equality of S-expression is determined by the LISP predicate equal.

However, in some cases in the following discussions, two m-expressions may be defined to be equivalent when $p[x_1; \ldots\ldots; x_n] \underset{w}{\equiv} q[x_1; \ldots\ldots; x_n]$ and $Dp = Dq$, and may be represented as $p[x_1; \ldots\ldots; x_n] = q[x_1; \ldots\ldots; x_n]$. Also, the equality based upon definition may be represented by the sign $\equiv$ to avoid confusion of equality of lists and equivalence of m-expressions.

〔Definition 1〕 The computation process is a LISP function in S-expression which is constructed from the following five primitive functions: CAR, CDR, CONS, ATOM, and EQ, using function composition, conditional expression, and recursion. Furthermore, it must not contain free variables and satisfy the following two conditions:

1) There is nothing in common between the λ variables and the <u>label</u> variables (i.e. function names).

2) The same <u>label</u> variable must not be used as the name for different functions.

〔Definition 2〕 The variables in a computation process are the variables in the outermost λ-expression contained in the given computation process.

Example: Computation process	Variables
(LAMBDA (X Y Z)(FN X Y Z))	X, Y, Z
(LABEL FN (LAMBDA (U V W) (G (CONS U V) W)))	U, V, W

The three restrictions in Definition 1 were adopted merely in order to simplify the discussion. Let it be noted that they can be removed easily.

〔Definition 3〕 If the computation process is represented as π and the list of initial values of the variables as args, then the evaluation of π at args is expressed as follows:

apply〔π; args; NIL〕

The sequence of variables in the computation process π is expressed by η. Any sub-sequence of η is expressed by η_1. After η_1 has been eliminated from η, the remaining sequence is expressed as η_2. Let the following be posited:

$$\eta_1 \equiv C_1, \ldots, C_m$$

$$\eta_2 \equiv R_1, \ldots, R_n$$

The symbols η_1' and η_2' are used to represent the sequences of values which are bounded to every variables of η_1 and η_2, respectively. $C_i (i=1, \ldots, m)$ is called the compile time variable (hereinafter abbreviated to cv), and $R_j (j=1, \ldots, n)$ is called the run time variable (herein-after abbreviated to rv). Furthermore, let the following also be posited:

$$cl_\pi \equiv (\eta_1)$$

$$rl_\pi \equiv (\eta_2)$$

$$cl'_\pi \equiv (\eta_1')$$

$$rl'_\pi \equiv (\eta_2')$$

〔Definition 4〕 The transformation given by α, the algorithm satisfying the following Equation (1), shall be called the partial evaluation of the computation process π at value (sequence of values) cl'_π with respect to variable (sequence of variables) cl_π, and α shall be called the partial evaluation algorithm.

$$\text{apply}[\alpha[\pi;\ rl_\pi;\ cl_\pi;\ cl'_\pi];\ rl'_\pi;\ \text{NIL}]$$

$$\underset{W}{\equiv}\ g[\pi;\ rl_\pi;\ cl_\pi;\ cl'_\pi;\ rl'_\pi;\ \text{NIL}] \text{-----------------} (1)$$

Here,

$$g[\pi;\ rl_\pi;\ cl_\pi;\ cl'_\pi;\ rl'_\pi;\ a]$$

$\equiv$ [atom[π] $\rightarrow$ g[eval[π; a]; rl_π; cl_π; cl'_π; rl'_π; a];

eq[car[π]; LAMBDA] $\rightarrow$ eval[caddr[π];

pairlis[cl_π; cl'_π; pairlis[rl_π; rl'_π; a]]];

eq[car[π]; LABEL] $\rightarrow$ g[caddr[π]; rl_π;

cl_π; cl'_π; rl'_π; append[list_1[cons[cadr[π];

caddr[π]]]; a]]],

list_1[x] $\equiv$ cons[x; NIL]

The right side of Equation (1) is the evaluation of after the values corresponding to each of the variables $C_1, \ldots, C_m, R_1, \ldots, R_n$ have been given. In other words, if the variables of π are arranged in a λ variable list in the order $C_1, \ldots, C_m, R_1, \ldots, R_n$, the right side of Equation (1) will be:

$$\mathrm{apply}[\pi;\ \mathrm{append}[\mathrm{cl'}_\pi;\ \mathrm{rl'}_\pi];\ \mathrm{NIL}]$$

Positing

$$\mathrm{val}[\pi;\ \mathrm{cl}_\pi;\ \mathrm{cl'}_\pi;\ \mathrm{rl}_\pi;\ \mathrm{rl'}_\pi]$$

$$\equiv g[\pi;\ \mathrm{rl}_\pi;\ \mathrm{cl}_\pi;\ \mathrm{cl'}_\pi;\ \mathrm{rl'}_\pi;\ \mathrm{NIL}]$$

let us rewrite Equation (1) as follows:

$$\mathrm{apply}[\alpha[\pi;\ \mathrm{rl}_\pi;\ \mathrm{cl}_\pi;\ \mathrm{cl'}_\pi];\ \mathrm{rl'}_\pi;\ \mathrm{NIL}]$$

$$\underset{w}{\equiv} \mathrm{val}[\pi;\ \mathrm{cl}_\pi;\ \mathrm{cl'}_\pi;\ \mathrm{rl}_\pi;\ \mathrm{rl'}_\pi] \text{-------------------} (2)$$

The trivial algorithm satisfying Equation (1) will substitute quoted values of the cv variables contained in π for corresponding variables. It would be useless to perform partial evaluation of π using this algorithm. Therefore, the algorithm sought in this paper will be a more complex one. It must satisfactorily satisfy the following two contradictory requirements.

d_1) It must be able to evaluate as many portions of π as possible which can be evaluated only by the constants contained in π and by the values of the cv. However,

d_2) As far as possible, it ought not to evaluate those portions which are actually not evaluated when the values of cv and rv are given to evaluate π.

(Note that the partial evaluation described in the Introduction refers to a special type of partial evaluation which satisfies only d_1, but not d_2.)

The purpose of requirement d_1 is the following. After partial evaluation of π has been completed, a new computation process is obtained; and the values of the rv are supplied, and it is evaluated. This requirement is for the purpose of reducing the time complexity in computation when this is done. The purpose of the second requirement is to eliminate wasteful processes during partial evaluation.

In the following are given intuitive explanations, using graphs, of the general concept of the partial evaluation algorithm given informally in Chapter 3 of this paper (hereinafter this algorithm is called α_1). In the ensuing graphs, the nodes (symbol o) indicate branching points depending upon truth-or-falsehood judgment.

The branches (arrows) indicate the computation processes and flow of computations not containing branching point. The leaves (symbol ●) indicate the termination of a computation process. Let us suppose that we have a computation process π represented graphically as in Fig. 1.

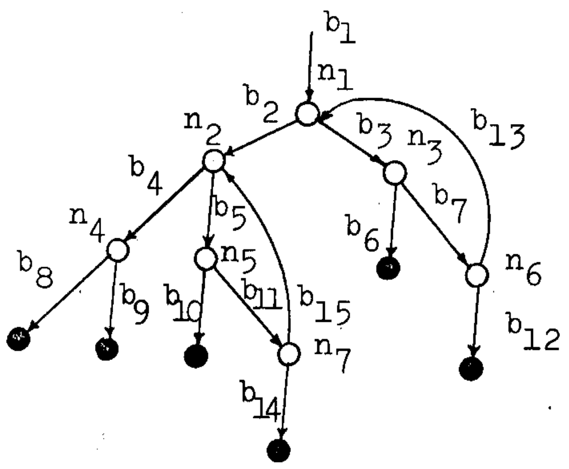


Fig. 1. Graphic representation of π

n_1, ..., n_7 are names given to the nodes.

b_1, ..., b_{15} are names given to the branches.

Let us suppose that the variables of π are C_1, ..., C_m, R_1, ..., R_n, and that π is to be subjected to partial evaluation at the value cl'_π with respect to C_1, ..., C_m. If the computation process beginning from branch b_i is expressed as π_i, then Fig. 1 contains fifteen computation processes: $\pi_1(=\pi)$, π_2, ..., π_{15}. Starting from

$$\mathrm{cl}\pi_1 \equiv \mathrm{cl}\,\pi$$

$$\mathrm{cl}'\pi_1 \equiv \mathrm{cl}'\pi$$

we proceed to define $\mathrm{cl}\pi_i$ and $\mathrm{cl}'\pi_i$ $(i = 1, \ldots, 15)$ in the following way.

$\mathrm{cl}\pi_i$: If the control is transfered to π_i from branch b_e, then the list of the variables of π_i of which corresponding arguments can be evaluated only by the constants and $\mathrm{cl}'\pi_e$.

$\mathrm{cl}'\pi_i$: The list of arguments corresponding to $\mathrm{cl}\pi_i$. (Note that $\mathrm{cl}\pi_i$ and $\mathrm{cl}'\pi_i$ may vary from computation to computation.)

Let us subject π to partial evaluation by means of operations i) and ii) described below. Here, we assume that

$$i: = 1;\ j(1): = j(2): = \ldots = j(15): = 1$$

i) In b_i, we evaluate the portions which can be evaluated by means of $\mathrm{cl}'\pi_i$ and the constants alone. We attach marks to them indicating that these portions have already been evaluated. The new branch obtained by this operation is called $b_i{}^{j(i)}$ and operation ii) is parformed. ($b_i{}^{j(i)}$ is a new computation process resulting from b_i, not a reformulation of b_i. Therefore, b_i is left in its previous form.)

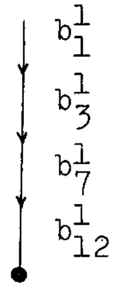
ii) Suppose that the process following b_i (in other words, the arrow-point of b_i) is judgment $n_{k(i)}$.

If the judgment of $n_{k(i)}$ can be performed by $cl'\pi_i$ and the constants alone, then one of the branches is selected, depending upon the result of the judgment (predicate). If, for instance, the branch is b_p, we carry out $i: = p;\ j(i): = j(i) + 1$; and execute operation i).

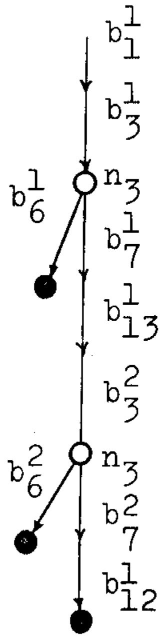
If the judgment of $n_{k(i)}$ cannot be performed by $cl'\pi_i$ and the constants alone, no judgment is performed. At each of the following two branches (b_p and b_q) the suffix (p or q) is assigned to i, j(i) is increased by 1, and operation i) is applied.

If b_i is followed by the sign ● indicating the termination of the computation process, partial evaluation is discontinued.

For example, let us suppose that it has been possible to perform the judgments of n_1, n_3, and n_6 in Fig. 1 in terms of $cl'\pi_i (i = 1, 3, 6)$ and the constants alone, and that branches b_3, b_7, and b_{12} have been selected on the basis of each of these judgments. In this case, π will be transformed into the following computation process on account of operations i) and ii).

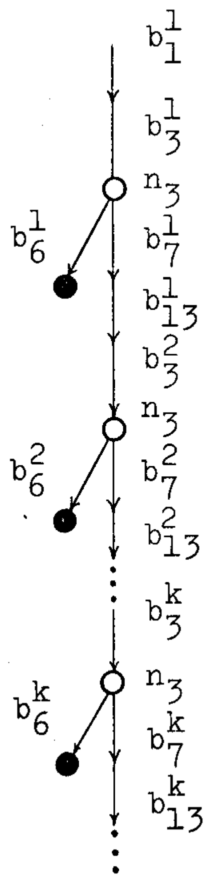


Let us consider a case in which it has been possible to perform the judgments of n_1 and n_6 by means of cl'π_i (i = 1, 6) and the constants alone, but in which it was not possible to perform the judgment of n_3 by means of cl'π_3 and the constants alone. Suppose that branch b_3 is always selected for n_1, and that for n_6, b_{13} is first selected, and b_{12} is next selected. In this case, π will be transformed into the following computation process.



Let it be noted that no consideration is paid to requirement d_2) in operations i) and ii).

In the example given above, if branch b_{13} is always selected for judgment n_6, operations i) and ii) will not come to an end. In other words, the following infinite sequence will be generated:

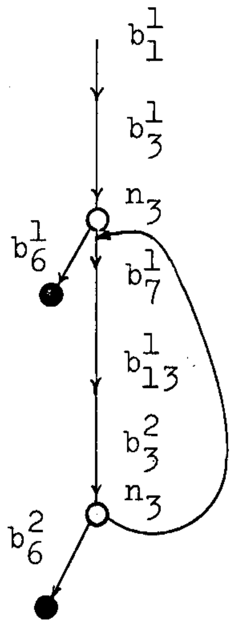


If the reason why branch b_{13} is always selected for judgment n_6 is the fact that the argument of judgment n_6 is constant or periodic, then it will be possible to generate a finite graph by adding the following operations to i) and ii).

ia) At the beginning of operation i), one generates $a_i^{j(i)}$, the name of $b_i^{j(i)}$, and puts the pair $cl'\pi_i$ and $a_i^{j(i)}$ at the list determined by π_i and $cl\pi_i$, for example $stk[\pi_i; cl\pi_i]$.

iia) Before proceeding from operation ii) to i), an investigation is made to see if the list $stk[\pi_p; cl\pi_p]$ contains a pair having $cl'\pi_p$ as its first term. If there is such a pair, an arrow is drawn at the corresponding $b_p^{j(p)}$, (which is found by its name $a_p^{j(p)}$) and the partial evaluation is discontinued. If there is no such pair, operation i) is performed.

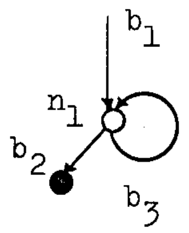
When this has been done, the computation process in the previous example will assume the form of a finite graph as shown in the following diagram.



α_1 is the algorithm in which ia) and iia) are added to operations i) and ii), and in which requirement d2) is also taken into consideration. However, the portions related to requirement d2) are not dealt with in this chapter.

The following is an example of a computation process in which α_1 does not terminate.

In the diagram below, suppose that judgment n_1 cannot be executed by means of cl'π_1 or cl'π_3 and the constants alone. At this time, even if the b_3 process is subjected to iterative partial evaluation, an infinite graph will still be generated as long as the same cl'π_3 is not produced. (That is, α_1 will not terminate.)



Even in cases when val(π; clπ; cl'π; rlπ; rl'π) terminates (that is, is defined), it does not necessarily follow that α_1(π; rlπ; clπ; cl'π) will terminate, and vice versa. These questions will be touched upon in the following chapter after α_1 has been described in detail.

3 Partial Evaluation Algorithm

Let us describe informally the algorithm α_1 for partial evaluation of the computation process (LISP S-expression). A formal description of this algorithm is given in the Appendix 1.

3.1 Outline of Function of α_1

In the case of α_1, the computation process, for example π_1, is analyzed by a method similar to the universal function apply. Evaluation is performed of the partial process of π_1 which can be evaluated by the value of cv_{π_1} and the constants alone. The portion which cannot be evaluated is copied unaltered, and a new computation process not containing cv_{π_1} is made up. For example, let us suppose that π_1 and its partial process π_2 are as follows:

$$\pi_1 \equiv \text{(LAMBDA (X Y Z) }(\underbrace{\text{(LAMBDA (U V) (CONS U V))}}_{\pi_2}\text{ X (CONS Y Z)))}$$

$cl\pi_1 \equiv$ (Y Z)

$cl'\pi_1 \equiv$ (A B)

$rl\pi_1 \equiv$ (X)

$rl'\pi_1 \equiv$ (C)

At this time, the task of α_1 will be to evaluate the X and the (CONS Y Z) of (π_2 X (CONS Y Z)). Since X is rv_{π_1}, it is left unmodified. Since Y and Z are cv_{π_1}, (CONS Y Z) is evaluated. X and (CONS Y Z) correspond to the π_2 variables U and V, respectively. Since the expression containing $rv\pi_1$ (X, in other words, the $rv\pi_1$ itself) corresponds to U, we regard U as $rv\pi_2$. Since the expression not containing $rv\pi_1$ corresponds to V, we regard V as $cv\pi_2$. Thus, we assume that

$$cl\pi_2 = (V)$$
$$cl'\pi_2 = ((A\cdot B))$$
$$rl\pi_2 = (U)$$

Thus, we make up (LABEL π^1_1 (LAMBDA (X) (π^1_2 X))). Where, π^1_2 is made up from π_2 in the following manner. The task of α_1 is to evaluate the (CONS U V) of π_2. Since U is $rv\pi_2$, it is left unmodified. Since V is $cv\pi_2$, QUOTE is added to its value. (When the function part is neither a λ-expression or a label expression, QUOTE is attached to the result of evaluation of the argument.) Thus, (LABEL π^1_2 (LAMBDA (U) (CONS U (QUOTE (A·B)))) is made up and used as π^1_2. That is:

π^1_1 = (LAMBDA (X) (π^1_2 X))
π^1_2 = (LAMBDA (U) (CONS U (QUOTE (A·B))))

At this time, the following relationship obviously applies:

apply[π_1; (C A B); NIL] = apply[π^1_1; (C); NIL]= (C·(A·B))

When the computation process π_3 is a label expression of the following type:

$\pi_3 \equiv$ (LABEL APPEND π_4)
$\pi_4 \equiv$ (LAMBDA (X Y) (COND ((NULL X) Y) (T (CONS (CAR X) (APPEND (CDR X) Y)))))

then

$$cl\pi_3 = cl\pi_4$$

Let us now assume that:

$$cl\pi_3 \equiv (X)$$

$$cl'\pi_3 \equiv ((A\ B))$$

$$rl\pi_3 \equiv (Y)$$

α_1 memorizes APPEND and π_4 as a pair. (This is the same as the treatment of LABEL in apply.) Thus, when the (NULL X) of π_4 is evaluated, since the value is NIL, the attempt will be made to evaluate (CONS (CAR X) (APPEND (CDR X) Y)). (CAR X) is evaluated, and its value is marked QUOTE. (CDR X) is evaluated, and (LABEL π_4^1 (LAMBDA (Y) (CONS (QUOTE A) (π_4^2 Y)))) is made up. Where, in the case of π_4^2, it is assumed that:

$$cl\pi_4 = (X)$$
$$cl'\pi_4 = ((B))$$

$$\mathrm{rl}_{\pi_4} = (\mathrm{Y})$$

and π_4 is subjected to partial evaluation in the same way as previously. That is,

$$\pi_4^2 = \text{(LAMBDA (Y) (CONS (QUOTE B) (}\pi_4^3\text{ Y)))}$$

Similarly, π_4^3 also is the result of partial evaluation of π_4 assuming that:

$$\mathrm{cl}_{\pi_4} = (\mathrm{X})$$
$$\mathrm{cl}'_{\pi_4} = (\mathrm{NIL})$$
$$\mathrm{rl}_{\pi_4} = (\mathrm{Y})$$

In other words, since (NULL X) is T, Y is selected, and we obtain:

$$\pi_4^3 = \text{(LAMBDA (Y) Y)}$$

At this point the partial evaluation of π_3 comes to an end.

Before performing the partial evaluation of a computation process, for example of π, α_1 will generate the name (i.e. the head location) π^1 of the new computation process to be obtained by partial evaluation, and will posit the pair of cl'_π and π^1 at the list determined by π and $\mathrm{cl}\pi$ -- for example, $\mathrm{stk}[\pi; \mathrm{cl}_\pi]$. For instance, in the example given above, let us now assume that:

$$\mathrm{cl}\pi_3 \equiv (\mathrm{Y})$$

$$\mathrm{cl}'\pi_3 \equiv ((\mathrm{C\ D}))$$

$$\mathrm{rl}\pi_3 \equiv (\mathrm{X})$$

α_1 will first generate the name π_4^1 and will put $(((\mathrm{C\ D}))\cdot\pi_4^1)$ at stk$[\pi_4; (\mathrm{Y})]$. Since (NULL X) cannot be evaluated unless the values of rvπ_3 are given, the following is made up:

```
(LABEL π¹₄ (LAMBDA (X) (COND ((NULL X) (QUOTE (C D)))
(T (CONS (CAR X) (π²₄ (CDR X)))))))
```

Here, π_4^2 is formed by partial evaluation of π_4, assuming that:

$$\mathrm{cl}\pi_4 = (\mathrm{Y})$$

$$\mathrm{cl}'\pi_4 = ((\mathrm{C\ D}))$$

$$\mathrm{rl}\pi_4 = (\mathrm{X})$$

α_1 investigates the list stk$[\pi_4; (\mathrm{Y})]$. Since $(((\mathrm{C\ D}))\cdot\pi_4^1)$ is already here, π_4^1 is used as the result of the partial evaluation of π_4. In other words, $\pi_4^2 = \pi_4^1$.

3.2 Informal Description of α_1

$$\alpha_1[\pi; \mathrm{rl}\pi; \mathrm{cl}\pi; \mathrm{cl}'\pi] \equiv \beta[\pi; \mathrm{rl}\pi; \mathrm{cl}\pi; \mathrm{cl}'\pi; \mathrm{NIL}; \mathrm{NIL}]$$

β has six variables: a_1, a_2, a_3, a_4, a_5 and a_6. As their values, they are given the computation process π, rl_π, cl_π, cl'_π, the list for processing of LABEL, and the list for memorizing the names once they have been generated. The functions of β are shown in terms of the following operations 1) through 5)

1) This operation is similar to the LISP interpreter apply.

1.1) If rl_π is NIL, that is if all of the variables of π are cv_π, then π is evaluated using cl'_π, and a no-argument function having the result as constant is made up.

$$\beta[\pi;\ \text{NIL};\ cl_\pi;\ cl'_\pi;\ a_5;\ a_6]$$
$$= \text{list } 3[\text{LAMBDA};\ \text{NIL};\ \text{list } 2[\text{QUOTE};\ g[\pi;\ \text{NIL};\ cl_\pi;\ cl'_\pi;\ \text{NIL};\ a_5]]]$$

Here, list n (n = 1, 2, 3, ...) is defined as the following function:

$$\text{list } n[x_1;\ \ldots;\ x_n] = \text{cons}[x_1;\ \text{cons}[x_2;\ \ldots;\ \text{cons}[x_n;\ \text{NIL}]]\ldots]$$

1.2) When π is an atom, it is evaluated in terms of a_5. And β is executed regarding the value of π as a new π.

$$\beta[\pi;\ rl_\pi;\ cl_\pi;\ cl'_\pi;\ a_5;\ a_6]$$
$$=\beta[\mathrm{eval}[\pi;\ a_5];\ rl_\pi;\ cl_\pi;\ cl'_\pi;\ a_5;\ a_6]$$

1.3) When π is a λ-expression, if on the list stk$[\pi;\ cl_\pi;\ a_6]$ there is a doted pair with cl'_π as its first term, then its second term will be taken as the value of β.

If this is not the case, a name for example π^1, is generated for the results of the partial evaluation, and they are put in the list stk$[\pi;\ cl_\pi;\ a_6]$ as a doted pair with cl'_π. Then the results of the processing of π in 2) and the following sections will be a label-body[7] of a label expression which has π^1 as its label variable.

1.4) If π is a label-expression, the label variable and the label-body are taken as a pair and put in list a_5; and the λ-body[8] of the label-body is processed by means of 1).

$$\beta[\pi;\ rl_\pi;\ cl_\pi;\ cl'_\pi;\ a_5;\ a_6]$$
$$=\beta[\mathrm{caddr}(\pi);\ rl_\pi;\ cl_\pi;\ cl'_\pi;\ \mathrm{append}[\mathrm{list}\ 1[\mathrm{cons}[\mathrm{cadr}(\pi);\ \mathrm{caddr}(\pi)]];\ a_5];\ a_6].$$

7) <label expression>::=(LABEL<label variable><label-body>)
<label-body>::=<λ-expression>

8) <λ-expression>::=(LAMBDA<λ variable><λ-body>)
<λ-body>::=<form>

2) The λ variable is changed into rl_π, and the λ-body is processed by means of 3).

$$\beta[\pi; rl\pi; cl\pi; cl'\pi; a_5; a_6]$$
$$= \text{list } 3[\text{LAMBDA}; rl\pi; \beta_3[\text{caddr}[\pi]; rl\pi; cl\pi; cl'\pi; a_5; a_6]]$$

Here, β_3 indicates processing at 3)

3) Here, form processing is performed by a method similar to <u>eval</u>. This is expressed as $\beta_3[\text{form}\pi; rl\pi; cl\pi; cl'\pi; a_5; a_6]$.

3.1) If form does not contain rv_π, cl'_π is used to evaluate form_π, and the results are used as the constants. (That is, QUOTE is added.)

$$\beta_3[\text{form}\pi; rl\pi; cl\pi; cl'\pi; a_5; a_6]$$
$$= \text{list } 2[\text{QUOTE}; \text{eval}[\text{form}_\pi; \text{pairlis}[cl\pi; cl'_\pi; a_5]]]$$

3.2) If $\text{form}\pi$ is $rv\pi$, the results will be $\text{form}\pi$ itself.

$$\beta_3[\text{form}_\pi; rl\pi; cl\pi; cl'\pi; a_5; a_6] = \text{form}\pi$$

3.3) If $\text{car}[\text{form}_\pi]$ is a primitive, each of the elements of $\text{cdr}[\text{form}_\pi]$ is subjected to processing at 3).

β_3[form$_\pi$; rl$_\pi$; cl$_\pi$; cl'$_\pi$; a$_5$; a$_6$]
= cons[car[form$_\pi$]; elist[cdr[form$_\pi$]; rl$_\pi$;
cl$_\pi$; cl'$_\pi$; a$_5$; a$_6$]].

Here, elist[y] will make a list of the elements of list y after subjecting them to processing at 3).

3.4) If form$_\pi$ is a conditional expression, cdr[form$_\pi$] is processed at 4).

3.5) If car[form$_\pi$] is a λ-expression, from among the λ variables, one assembles together those corresponding to the elements of cdr[form$_\pi$] not containing rv$_\pi$. With these the cl of car[form$_\pi$] is made up. The remaining variables are gathered together to make up rl. Among the elements of cdr[form$_\pi$], those not containing rv$_\pi$ are evaluated using cl'$_\pi$. A list is made up, with these as its elements, and is used as the cl'$_\pi$ of car[form$_\pi$].

After this has been done, car[form$_\pi$] is processed by 1), and the result is consed with the list of the elements of cdr[form$_\pi$] containing rv$_\pi$, and subjected to processing at 3).

3.6) When car[form$_\pi$] is a label-expression, the same processing as in 3.5) is performed. However, instead of subjecting car[form$_\pi$] to the processing at 1), the dotted pair of the label variables and the label-body are put at a_5, and caddr[form$_\pi$] is subjected to processing at 1).

3.7) If car[form$_\pi$] is an atom, that is, a label variable, its definition expression is extracted from a_5, and this is used as car[form$_\pi$] and subjected to the processing at 3).

$$\beta_3[\mathrm{form}_\pi;\ \mathrm{rl}_\pi;\ \mathrm{cl}_\pi;\ \mathrm{cl}'_\pi;\ a_5;\ a_6]$$
$$= \beta_3[\mathrm{cons}[\mathrm{eval}[\mathrm{car}[\mathrm{form}_\pi];\ a_5];\ \mathrm{cdr}[\mathrm{form}_\pi]];\ \mathrm{rl}_\pi;\ \mathrm{cl}_\pi;\ \mathrm{cl}'_\pi;\ a_5;\ a_6]$$

4) At this point, the conditional expression is processed. This operation represented in the following

$$\beta_4[x_\pi\ \mathrm{rl}_\pi;\ \mathrm{cl}_\pi;\ \mathrm{cl}'_\pi;\ a_5;\ a_6].$$

Here, x_π assumes the appearance of:

$$((p_1,\ v_1),\ \ldots,\ (p_k,\ v_k))$$

4.1) When the leftmost predicate part of x_π (for instance, p_i) does not contain rv_π, p_i is evaluated using cl'_π. If its value is T, the corresponding value part v_i is subjected to processing at 3).

$$\beta_4[x_\pi;\ \mathrm{rl}_\pi;\ \mathrm{cl}_\pi;\ \mathrm{cl}'_\pi;\ a_5;\ a_6] = \beta_3[v_i;\ \mathrm{rl}_\pi;\ \mathrm{cl}_\pi;\ \mathrm{cl}'_\pi;\ a_5;\ a_6]$$

If its value is F, x_π is replaced by cdr$[x_\pi]$, and the processing of 4.1) is repeated.

If p_i contains rv_π, the processing of 4.2) is performed.

4.2) A conditional expression is made up. Its predicate parts and value parts consist of the predicate parts p_j and value parts v_j of x_π which have been subjected to processing of 5).

$$\beta_4[x_\pi; rl_\pi; cl_\pi; cl'_\pi; a_5; a_6]$$
$$= \text{listn}[\text{COND}; \text{list 2}[\beta_5[p_i; rl_\pi; cl_\pi; cl'_\pi; a_5; a_6]; \beta_5[v_i; rl_\pi; cl_\pi; cl'_\pi; a_5; a_6]]; \ldots; \text{list 2}[\beta_5[p_k; rl_\pi; cl_\pi; cl'_\pi; a_5; a_6]; \beta_5[v_k; rl_\pi; cl_\pi; cl'_\pi; a_5; a_6]]].$$

Here, $n = k - i + 1$, and β_5 indicates processing by 5).

5) Here, the predicate part containing rv_π, and the predicate parts and value parts appearing to the right of the foregoing are processed. This processing is represented in the following terms:

$$\beta_5[\text{form}_\pi; rl_\pi; cl_\pi; cl'_\pi; a_5; a_6]$$

5.1) In cases when form does not contain rv_π:

5.1.1) • If form_π is an atom, QUOTE is attached to its value.

5.1.2) If car[form$_\pi$] is QUOTE, it is left unmodified.

5.1.3) If form$_\pi$ is a conditional expression, it is processed by 5.2).

5.1.4) Otherwise, the elements of cdr[form$_\pi$] are subjected to the processing of 5).

5.2) When form$_\pi$ is a conditional expression. The following is posited: cdr[form$_\pi$] = x_π = ((p_1, v_1), ..., (p_k, v_k)).

5.2.1) When the leftmost predicate part of x_π, for instance p_i, does not contain rv$_\pi$, p_i is evaluated by using cl'$_\pi$. If its value is T, the corresponding value part v_i is subjected to processing by 5).

If the value is F, x_π is replaced by cdr[x_π], and processing is performed by 5.2.1).

If p_i contains rv$_\pi$, the processing in 4.2) is performed. The processing 5.2) is one which performs processing of 5) instead of giving the processing of 3) to the value part corresponding to the predicate part which first becomes T in 4.1).

5.3) Otherwise, form$_\pi$ is subjected to the processing of 3).

Partial evaluation algorithm α_1 was described informally in terms of the operations 1) through 5) given above. As for the partial processes concerning which it is not known whether evaluation is actually possible or not, that is the partial processes succeeding a judgment containing rv, if these processes do not contain rv, they are not evaluated, but the values of cv are merely assigned (Process 5.1). However, if such a partial process is a conditional expression, the predicate part is evaluated and the value part is extracted. After this, the values are assigned to the cv contained in the value part (Process 5.2.1). If the partial process contains rv, partial evaluation is performed by applying β_3 recursively (Process 5.3).

Process 5.1 satisfies requirement d2 in Chapter 2, which called for avoidance of wasteful evaluations. However, on the other hand, it goes against requirement d1, which calls for the evaluation of portions where evaluation is possible even when the value of rv is unknown. Process 5.3 satisfies requirement d1 but goes against requirement d2. Process 5.2.1 goes against requirement d2 in that the predicate part is evaluated, but it goes against requirement d1 in that the value part is not evaluated.

The problem is a question of balancing the different requirements against each other, that is, of deciding exactly how much weight is to be given to each requirement in any given case. In α_1 wasteful computation is avoided whenever possible, but those judgments are executed which can be performed even when the value of rv is unknown. In this way, an attempt is made to reduce the number of judgments contained in the computation processes obtained by means of partial evaluation.

3.3 Halting Problem of α_1

As is obvious from Processes 1.1 and 3.1 (i.e. α_1 contains apply and eval), the halting problem (i.e., whether α_1 will or will not terminate with respect to any given argument) is undecidable. Even though val $[\pi;\ cl_\pi;\ cl'_\pi;\ rl_\pi;\ rl'_\pi]$ should terminate, it does not necessarily follow that α_1 $[\pi;\ rl_\pi;\ cl_\pi;\ cl'_\pi]$ will likewise terminate. This is so in view of processes 4.2, 5.2 and 5.3. On the contrary, even though val $[\pi;\ cl_\pi;\ cl'_\pi;\ rl_\pi;\ rl'_\pi]$ does not terminate, it is clear from processes 1.3 and 3.5 that α_1 $[\pi;\ rl_\pi;\ cl_\pi;\ cl'_\pi]$ may sometimes terminate.

4 Compiler Generation from Interpreter

The interpreter of the programming language is a computation process in which a sentence belonging to the language is regarded as a representation of the computation process. Values of variables (the so-called state vectors [4]) are assigned, and the sentence is evaluated. Here let us represent the interpreter by int and the variables by L, X, Y_1, ..., Yn. These variables indicate the following:

L : Sentence belonging to the language (program).

X : List of values of variables contained in the sentence (state vector).

Y_1, ..., Yn : States of the tables or stacks used to analyze the sentence syntax or to evaluate the sentence.

If the values of L, X, Y_1, ..., Yn are ℓ, x, y_1, ..., y_n, respectively, then the evaluation of sentence ℓ by means of int will be:

$$\text{apply[int; list}[\ell;\ x;\ y_1;\ \ldots,\ y_n];\ \text{NIL]}$$

Some of the variables among Y_1, ..., Yn indicate the states of tables and stacks which are used only when analyzing sentence (program) syntax. These are represented as Y_{i1}, ..., Y_{ip}, and the remaining variables are represented as Y_{j1}, ..., Y_{jq}. The following are posited:

$$\mathrm{cl}_{\mathrm{int}} \equiv (L, Y_{i1}, \ldots, Y_{ip})$$

$$\mathrm{cl'}_{\mathrm{int}} \equiv \mathrm{list}[\ell; y_{i1}; \ldots; y_{ip}]$$

$$\mathrm{rl}_{\mathrm{int}} \equiv (X, Y_{j1}, \ldots, Y_{jq})$$

$$\mathrm{rl'}_{\mathrm{int}} \equiv \mathrm{list}[x; y_{j1}; \ldots; y_{jq}]$$

Then, from the definition of <u>val</u> in Chapter 2, we obtain:

$$\begin{aligned} &\mathrm{apply}[\mathrm{int}; \mathrm{list}[\ell; x; y_1; \ldots; y_n]; \mathrm{NIL}] \\ &= \mathrm{val}[\mathrm{int}; \mathrm{cl}_{\mathrm{int}}; \mathrm{cl'}_{\mathrm{int}}; \mathrm{rl}_{\mathrm{int}}; \mathrm{rl'}_{\mathrm{int}}] \qquad \text{----} \ (3) \end{aligned}$$

Therefore, if <u>int</u> is partially evaluated at $\mathrm{cl'}_{\mathrm{int}}$ with respect to $\mathrm{cl}_{\mathrm{int}}$, the following is obtained:

$$\begin{aligned} &\mathrm{apply}[\alpha_1[\mathrm{int}; \mathrm{rl}_{\mathrm{int}}; \mathrm{cl}_{\mathrm{int}}; \mathrm{cl'}_{\mathrm{int}}]; \mathrm{rl'}_{\mathrm{int}}; \\ &\mathrm{NIL}] \underset{w}{\equiv} \mathrm{apply}[\mathrm{int}; \mathrm{list}[\ell; x; y_1; \ldots; y_n]; \mathrm{NIL}] \ \text{--} \ (4) \end{aligned}$$

When α_1 has been described in terms of a LISP m-expression, and when this has been transformed into an S-expression (for the algorithm for this transformation, refer to Reference 1), the result is posited as α_1^*. The variables of α_1^* are to be A_1, A_2, A_3 and A_4 (see Appendix 1).

The following are posited:

$$\mathrm{cl}_{\alpha_1^*} \equiv (A_1, A_2, A_3)$$

$$\mathrm{cl'}_{\alpha_1^*} \equiv \mathrm{list}[\mathrm{int}; \mathrm{rl}_{\mathrm{int}}; \mathrm{cl}_{\mathrm{int}}]$$

$$\text{rl}\,\alpha_1{}^* \equiv (A_4)$$

$$\text{rl}'\alpha_1{}^* \equiv \text{list}[\text{cl}'_{\text{int}}]$$

Then,

$$\alpha_1[\text{int};\ \text{rl}_{\text{int}};\ \text{cl}_{\text{int}};\ \text{cl}'_{\text{int}}]$$

$$= \text{apply}[\alpha_1{}^*;\ \text{list}[\text{int};\ \text{rl}_{\text{int}};\ \text{cl}_{\text{int}};\ \text{cl}'_{\text{int}}];\ \text{NIL}]$$

(from the definition of $\alpha_1{}^*$)

$$= \text{val}\ [\alpha_1{}^*;\ \text{cl}_{\alpha_1*};\ \text{cl}'_{\alpha_1*};\ \text{rl}_{\alpha_1*};\ \text{rl}'_{\alpha_1*}]$$

(from the definition of <u>val</u>)

$$\overset{=}{\text{w}}\ \text{apply}[\alpha_1[\alpha_1{}^*;\ \text{rl}_{\alpha_1*};\ \text{cl}_{\alpha_1*};\ \text{cl}'_{\alpha_1*}];\ \text{rl}'_{\alpha_1*};\ \text{NIL}]$$

(from the definition of α_1)

Then let us adopt the following definition:

$$\text{comp}^* \equiv \alpha_1[\alpha_1{}^*;\ \text{rl}_{\alpha_1*};\ \text{cl}_{\alpha_1*};\ \text{cl}'_{\alpha_1*}] \qquad \text{----}\ (5)$$

(Let it be noted that the value of the right side of Equation (5) is an S-expression.)

In this case, the following equation will apply:

$$\alpha_1[\text{int};\ \text{rl}_{\text{int}};\ \text{cl}_{\text{int}};\ \text{cl}'_{\text{int}}]\ \overset{=}{\text{w}}\ \text{apply}$$

$$[\text{comp}^*;\ \text{list}\ [\text{cl}'_{\text{int}}];\ \text{NIL}] \qquad \text{------}\ (6)$$

From (4) and (6), the following will be obtained:

$$\mathrm{apply}[\mathrm{apply}[\mathrm{comp}^*;\ \mathrm{list}[\mathrm{cl}'_{\mathrm{int}}];\ \mathrm{NIL}];\ \mathrm{rl}'_{\mathrm{int}};\ \mathrm{NIL}] \overset{=}{\mathrm{w}} \mathrm{apply}[\mathrm{int};\ \mathrm{list}[\ell;\ x;\ y_1;\ \ldots;\ y_n];\ \mathrm{NIL}] \quad \text{---}\ (7)$$

On account of the nature of α_1 and of Equation (6), in the evaluation of comp* computations concerning $\mathrm{cl}'_{\mathrm{int}}$ are performed; that is, the sentence (program) is subjected to syntax analysis. The computation process obtained by the evaluation of comp* satisfies Equation (7). Therefore, comp* can be regarded as the compiler performing syntax analysis of the program and translating it into the computation process. (Refer to Reference 4 for the equation indicating the correctness of the compiler. apply corresponds to machine.)

From the preceding discussion, the following characteristics are obtained concerning the relationship between the compiler and the interpreter.

1) The compiler is generated by giving the interpreters int, $\mathrm{rl}_{\mathrm{int}}$ and $\mathrm{cl}_{\mathrm{int}}$ and partially evaluating the partial evaluation algorithm $\alpha_1{}^*$. (From Equation 5)

2) When the compiled computation process is evaluated, computations concerning $\mathrm{rl}'_{\mathrm{int}}$ are performed, and computations concerning $\mathrm{cl}'_{\mathrm{int}}$, that is syntax analysis of the program, are either not performed at all, or are very few, if performed at all. Therefore, when performing iterative

computations with a fixed cl'_{int} (that is, with a fixed program) and varying rl'_{int} alone, or when iterations such as loops or recursive calls occur due to the execution of the program, there are many cases in which it is more advantageous to execute the program after first compiling. However, in cases when iterative computations are not performed, and when the program does not include any iterations such as loops or recursive calls, it is more advantageous to perform evaluation with the interpreter rather than to execute the program after compiling. (From Equation 7)

3) Even when the interpreter <u>int</u>, cl_{int}, and rl_{int} are given, it is not necessarily true that the compiler can be generated by using α_1. This is so because the right side of Equation (5) is not necessarily defined (i.e. α_1 does not always terminate) with respect to all $cl'_{\alpha_1 *}$.

In view of characteristic 3), the following problem arises: For interpreters of what language can compilers be made up by means of partial evaluation? Using the LISP system for HITAC 5020, we ascertained that compilers for a simple ALGOL-like language having assignment statements, conditional statements, go to statements, and block structures could be made by this method. (See Appendix 2.) However, this question will not be dealt with here in any further detail.

5 Conclusion

Partial evaluation of the computation processes as described in this paper is a new theory concerning computation methods. It is not as yet sufficiently formulated and has not yet emerged from the stage of an idea. The tasks left for the future are the formulation and application of this theory.

Formulation ought to be carried out mathematically as a branch of the science of computation. Formulation includes the formalisms for describing computation and storage as well as the problem of equivalence of computation.

As for applications, it is anticipated that the theory is capable of wide application to information processing in general on the basis of partial information. The first example is that taken up in this paper, the method of compiler generation, which can be applied to compiler-compiler.

It is thought that formulation and application can be carried out on the basis of the idea described in this paper.

In conclusion, the writer respectfully expresses his gratitude to Mr. Kazuma Yoshimura of the Central Research Laboratory, HITACHI, LTD., who kindly discussed this research from its earliest stages and pointed out a number of errors. The writer also thanks Dr. Shozo Shimada, of the same Laboratory, who took an interest in this research and provided the writer with helpful suggestions.

References

1) McCarthy, J., et al., LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, Massachusetts, 1962.

2) Futamura, Y., Concerning induction for proving computation processes, unpublished.

3) McCarthy, J., A basis for a mathematical theory of computation, in Computer Programming and Formal Systems, North-Holland, Amsterdam, 1963, 33-70.

4) McCarthy, J., Towards a mathematical science of computation, Proc. IFIP Congr. 62, North-Holland, Amsterdam, 1962, 21-28.

Appendix 1. Formal Description of α_1

The partial evaluation algorithm α_1, which was described informally in Chapter 3, is here described using the LISP m-expression. Functions β and β_1 described below correspond to operation 1 in Chapter 3, and β_2, β_3, β_4 and β_5 each correspond to operation 2, 3, 4, and 5, respectively. The other auxiliary functions, except those which are trivial, have been annotated.

nrv[a_1; a_2] is a predicate taking value T when form a_1 does not contain rv, that is the element of a_2, and taking value F when form a_1 does contain rv.

```
     nrv[a_1; a_2]
  ≡ [atom[a_1]→not[member[a_1; a_2]];
    eq[car[a_1]; QUOTE]→T;
    eq[car[a_1]; COND]→nrv1[cdr[a_1]; a_2];
    T → nrv2[cdr[a_1]; a_2]]
    nrv1[x; a_2]
  ≡ [null[x]→T;
    and[nrv[caar[x]; a_2]; nrv[cadar[x];
    a_2]]→nrv1[cdr[x]; a_2];
    T→NIL]
    nrv2[x; a_2]
  ≡ [null[x]→T;
  • nrv[car[x]; a_2]→nrv2[cdr[x]; a_2];
```

T → NIL]

$\alpha_1[a_1;\ a_2;\ a_3;\ a_4] \equiv \beta[a_1;\ a_2;\ a_3;\ a_4;\ NIL;\ NIL]$

$\beta[a_1;\ a_2;\ a_3;\ a_4;\ a_5;\ a_6]$

≡ [null[a_2] → list3[LAMBDA; NIL; list2[QUOTE;

g[a_1; NIL; a_3; a_4; NIL; a_5]]];

atom[a_1] → β[eval[a_1; a_5]; a_2; a_3; a_4; a_5; a_6];

eq[car[a_1]; LAMBDA] → β_1[stk[a_1; a_3; a_6];

gensym[ ]; a_1; a_2; a_3; a_4; a_5; a_6];

eq[car[a_1]; LABEL] → β[caddr[a_1]; a_2; a_3; a_4;

append[list1[cons[cadr[a_1]; caddr[a_1]]];

a_5]; a_6]]

The name of the value of β is generated by gensym[ ].

$\beta_1[x;\ y;\ a_1;\ a_2;\ a_3;\ a_4;\ a_5;\ a_6]$

≡ [null[x] → list3[LABEL; y; β_2[a_1; a_2; a_3; a_4; a_5;

cons[cons[cons[a_1; a_3]; list1[cons[a_4; y]]];

a_6]]];

null[assoc[a_4; cdr[x]]] → list3[LABEL;

y; β_2[a_1; a_2; a_3; a_4 a_5; cons[append[x; list1[cons

[a_4; y]]]; delete[x; a_6]]];

T → cdr[assoc[a_4; cdr[x]]]]

delete[x; y]

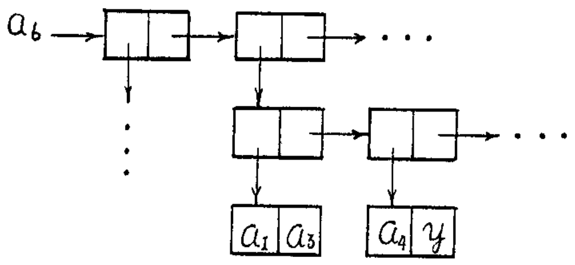
≡ [null[y] → NIL;

eq[x; car[y]] → cdr[y];

T → cons[car[y]; delete[x; cdr[y]]]]

$stk[a_1; a_3; a_6] \equiv assoc[cons[a_1; a_3]; a_6]$

The structure of a_6 will be as follows



$\beta_2[a_1; a_2; a_3; a_4; a_5; a_6]$

$\equiv list3[LAMBDA; a_2; \beta_3[caddr[a_1]; a_2; a_3; a_4; a_5; a_6]]$

$\beta_3[a_1; a_2; a_3; a_4; a_5; a_6]$

$\equiv [nrv[a_1; a_2] \rightarrow list2[QUOTE; eval[a_1; pairlis$
$[a_3; a_4; a_5]]];$

$member[a_1; a_2] \rightarrow a_1;$

$or\ 5[eq[car[a_1]; CAR];$

$eq[car[a_1]; CDR];$

$eq[car[a_1]; CONS];$

$eq[car[a_1]; EQ];$

$eq[car[a_1]; ATOM]; \rightarrow cons[car[a_1];$
$rlist[cdr[a_1]; a_2; a_3; a_4; a_5; a_6]];$

$eq[car[a_1]; COND] \rightarrow \beta_4[cdr[a_1]; a_2; a_3; a_4;$
$a_5; a_6];$

$eq[caar[a_1]; LAMBDA] \rightarrow cons[src[car[a_1];$
$cadar[a_1]; cdr[a_1]; a_2; a_3; a_4; a_5; a_6; a_5];$

```
rlist[rlist_1[cdr[a_1]; a_2]; a_2; a_3; a_4; a_5; a_6]];
      eq[caar[a_1]; LABEL]→ cons[src[caddar
[a_1]; cadaddar[a_1]; cdr[a_1]; a_2; a_3; a_4; append
[list1[cons[cadar[a_1]; caddar[a_1]]]; a_5]; a_6;
a_5]; rlist[rlist1[cdr[a_1]; a_2]; a_2; a_3; a_4; a_5;
a_6]];
      atom[car[a_1]] → β_3[cons[eval[car[a_1]; a_5];
cdr[a_1]]; a_2; a_3; a_4; a_5; a_6]]
```

Where,

$$\text{or } 5[x_1;\ x_2;\ \ldots;\ x_5] \equiv \text{or } [x_1;\ x_2;\ \ldots;\ x_5].$$

As for rlist $[x; a_2; a_3; a_4; a_5; a_6]$, a list will be compiled having as its elements the elements of list x to which β_3 has been applied.

```
   rlist[x; a_2; a_3; a_4; a_5; a_6]
≡ [null[x] → NIL;
   T → cons[β_3[car[x]; a_2; a_3; a_4; a_5; a_6];
   rlist[cdr[x]; a_2; a_3; a_4; a_5; a_6]]]
```

As for rlist_1 $[x; a_2]$, a list will be compiled consisting only of elements from list x which contain rv (that is, elements of a_2).

```
   rlist_1 [x; a_2]
≡ [null[x] → NIL;
   ·nrv[car[x]; a_2] → rlist_1[cdr[x]; a_2];
T → cons[car[x]; rlist_1[cdr[x]; a_2]]]
```

As for src[x; y; z; a_2; a_3; a_4; a_5; a_6; a_7], first the cl_x, cl'_x and rl_x relating the λ expression x are compiled. Next, x is processed by β.

```
    src[x; y; z; a_2; a_3; a_4; a_5; a_6; a_7]
  ≡ β[x; rl_1[z; y; a_2]; cl_1[z; y; a_2]; evlis[clist
    [z; a_2]; pairlis[a_3; a_4; a_7]]; a_5; a_6]
```

As for rl_1 [z; y; a_2] a list will be compiled consisting only of elements of y corresponding to elements of z containing elements of a_2.

```
    rl_1[z; y; a_2]
  ≡ [null[z] → NIL;
    nrv[car[z]; a_2] → rl_1[cdr[z]; cdr[y]; a_2];
    T → cons[car[y]; rl_1[cdr[z]; cdr[y]; a_2]]]
```

As for cl_1 [z; y; a_2] a list will be compiled consisting only of elements of y corresponding to elements of z not containing elements of a_2.

```
    cl_1[z; y; a_2]
  ≡ [null[z] → NIL;
    nrv[car[z]; a_2] → cons[car[y]; cl_1[cdr[z];
    cdr[y]; a_2]];
    T → cl_1[cdr[z]; cdr[y]; a_2]]
```

As for clist〔z; a_2〕 a list will be compiled consisting only of elements of z not containing elements of a_2.

```
        clist〔z; a_2〕
     ≡  〔null〔z〕→ NIL;
        nrv〔car〔z〕; a_2〕→ cons〔car〔z〕; clist〔cdr
        〔z〕; a_2〕〕;
        T → clist〔cdr〔z〕; a_2 〕〕
        β_4〔x; a_2; a_3; a_4; a_5; a_6〕
     ≡  〔nrv〔caar〔x〕; a_2〕→ 〔eval〔caar〔x〕; pairlis
        〔a_3; a_4; a_5 〕〕→β_3〔cadar〔x〕; a_2; a_3; a_4; a_5; a_6〕;
        T→ β_4〔cdr〔x〕; a_2; a_3; a_4; a_5; a_6〕〕;
        T→  cons〔COND; cond_1〔x; a_2; a_3; a_4; a_5; a_6〕〕〕
        cond_1〔x; a_2; a_3; a_4; a_5; a_6 〕
     ≡  〔null〔cdr〔x〕〕→ list 1〔list 2 〔β_5〔caar〔x〕; a_2;
        a_3; a_4; a_5; a_6〕; β_5〔cadar〔x〕; a_2; a_3; a_4; a_5; a_6〕〕〕;
        T → cons〔list 2 〔β_5〔caar〔x〕; a_2; a_3; a_4; a_5; a_6〕;
        β_5〔cadar〔x〕; a_2; a_3; a_4; a_5; a_6〕〕; cond_1〔cdr〔x〕;
        a_2; a_3; a_4; a_5; a_6 〕〕〕
        β_5〔x; a_2; a_3; a_4; a_5; a_6〕
     ≡ 〔nrv〔x; a_2〕→
        〔atom〔x〕→ list 2 〔QUOTE; eval〔 x; pairlis〔a_3;
        a_4; a_5〕〕〕;
              eq〔car〔x〕; QUOTE 〕→ x;
              eq〔car〔x〕; COND〕→ cond_2〔cdr〔x〕; a_2; a_3;
        a_4; a_5; a_6 〕;
```

```
T → cons[car[x]; cevlis[cdr[x]; a_2; a_3;
a_4; a_5; a_6]]];
     eq[car[x]; COND] → cond_2[cdr[x]; a_2; a_3;
a_4; a_5; a_6];
T → β_3[x; a_2; a_3; a_4; a_5; a_6]]

cond_2[y; a_2; a_3; a_4; a_5; a_6]
≡ [nrv[caar[y]; a_2] → [eval[caar[y]; pairlis
[a_3; a_4; a_5]] → β_5[cadar[y]; a_2; a_3; a_4; a_5; a_6];
T → cond_2[cdr[y]; a_2; a_3; a_4; a_5; a_6]];
T → cons[COND; cond_1[y; a_2; a_3; a_4; a_5; a_6]]]
```

As for cevlis[y; a_2; a_3; a_4; a_5; a_6], a list will be compiled having as its elements the elements of list y to which β_5 has been applied.

```
cevlis[y; a_2; a_3; a_4; a_5; a_6]
≡ [null[y] → NIL;
T → cons[β_5[car[y]; a_2; a_3; a_4; a_5; a_6];
cevlis[cdr[y]; a_2; a_3; a_4; a_5; a_6]]]
```

Appendix 2. Example of Compiler Generation

A simple ALGOL-like language containing assignment statements, conditional statements, go to statements, and block structures was described by the interpreter. LISP for HITAC 5020 computer was actually used to ascertain that it is possible to make up compilers from such interpreters by the method described in Chapter 4, using the partial evaluation algorithm α_1. In other words, it was ascertained that

$$\alpha_1 [\alpha_1^*; rl_{\alpha_1^*}; cl_{\alpha_1^*}; cl'_{\alpha_1^*}]$$

will terminate. The description here pertains to progf*, the interpreter used at that time.

The program features in LISP 1.5 are the functions which make it possible to write ALGOL-like programs containing assignment statements, conditional statements, go to statements, and block structures. The functions of the program features were imposed restrictions on the following two points, and progf* was used as the interpreter.

1) SETQ, SET, RETURN and PROG (block) can be written only as the top level of the program or as the value part of the top-level conditional statement.

2) It is impossible to go outside of the block by a go to

statement.

Even in this case, programs of the following type can be written.

```
     ((U V)
      (SETQ U N)
L1    (COND ((NULL U) (RETURN V)))
      (SETQ V (CONS (CAR U) V))
      (SETQ U (CDR U))
      (GO L1))
```

The program is represented as x, and a is used to represent the list having as its elements the dotted pairs of variables and their values. When the program given above is taken as x, if values (A, B, C) are given to N, then a = ((N·(A, B, C))).

progf[x; a] ≡ pfeval[cdr[x]; ppair[car [x]; a]; golist[cdr[x]; NIL]].

Where, in ppair[y; a], the dotted pairs consisting of the elements in list y and NIL are put in list a. In the case of the example given above,

ppair[car[x]; a] = ((U·NIL)(V·NIL) (N·(A, B, C))).

In golist[y; g], the dotted pairs of the labels (top level atoms) appearing in list y of the statements and the subsequently ensuing lists of the statements are put in list g.

```
      pfeval[x; a; g]
    ≡ [null[x] → NIL;
      atom[car[x]] → pfeval[cdr[x]; a; g];
           eq[caar[x]; GO] → pfeval[cdr[assoc[cadar
      [x]; g]]; a; g];
           eq[caar[x]; COND] → pfcond[cons[cdar[x];
      cdr[x]]; a; g];
           eq[caar[x]; SETQ] → prog2[rplacd[assoc
      [cadar[x]; a]; eval[caddar[x]; a]]; pfeval
      [cdr[x]; a; g]];
           eq[caar[x]; SET] → prog2[rplacd[assoc
      [eval[cadar[x]; a]; a]; eval[caddar[x]; a]];
      pfeval[cdr[x]; a; g]];
           eq[caar[x]; RETURN] → eval[cadar[x]; a];
           eq[caar[x]; PROG] → prog2[progf[cdar
      [x]; a]; pfeval[cdr[x]; a; g]];
      T → prog2[eval[car[x]; a]; pfeval[cdr
      [x]; a; g]]]
```

In pfcond[y; a; g], the top level conditional expression of the program is processed.

```
  pfcond[y; a; g]
≡ [null[car[y]] → pfeval[cdr[y]; a; g];
  eval [caaar[y]; a] →
  [ eq[caadaar[y]; GO] → pfeval[cdr[assoc
  [ cadadaar[y]; g]]; a; g];
      [eq[caadaar[y]; SETQ] → prog2[rplacd
  [assoc[cadadaar[y]; a]; eval[caddadaar [y]; a]];
  pfeval[cdr[y]; a; g]];
       eq[caadaar[y]; SET] → prog2[rplacd
  [assoc[eval[cadadaar[y]; a]; a]; eval[caddadaar
  [y]; a]]; pfeval[cdr[y]; a; g]];
       eq[ caadaar[y]; RETURN ] → eval [cadadaar
  [y]; a];
       eq[caadaar[y]; PROG] → prog2[progf
  [ cdadaar[y]; a]; pfeval[ cdr[y]; a; g]];
       T→  prog2[eval[cadaar[y]; a]; pfeval
  [cdr[y]; a; g]]];
       T → pfcond[cons[cdar [y]; cdr[y ]]; a; g]]
```

The progf in the m-expression given above, when transformed into an S-expression, is progf*.

The following are posited:

$rl_{\alpha_1^*} = (A_4)$

$cl_{\alpha_1^*} = (A_1, A_2, A_3)$

$$\text{cl}'\alpha_{1^*} = \text{list}[\text{progf}^*;\ (\text{A});\ (\text{X})]$$

It was ascertained that the partial evaluation of $\alpha_1{}^*$ terminated.