PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

PROGRAM EVALUATION AND GENERALIZED PARTIAL COMPUTATION

Yoshihiko Futamura*

Center for Research in Computing Technology
Aiken Computation Laboratory, Harvard University
Cambridge, MA02138, USA

ABSTRACT

Generalized Partial Computation (GPC) is a program optimization principle based on partial computation and theorem proving. Conventional partial computation methods (or partial evaluators) explicitly make use of only given parameter values to partially evaluate programs. However, GPC explicitly utilizes not only given values but also the following information:

1. Logical structure of a program to be partially evaluated.
2. Abstract data type of a programming language.

GPC is new (proposed in 1987) and even conventional partial computation is not well understood by the computer science society. This paper discusses (1) interesting properties of partial computation, (2) differences between program evaluators, partial evaluators and generalized partial evaluators and (3) principles and applications of GPC.

1 INTRODUCTION

Generalized Partial Computation (GPC)[FN88] is an extended idea of partial computation. Partial computation is a systematic method of generating an efficient program based on a given program and a part of its data. Partial computation has been considered in the following way with this kind of program generation [Fut71]:

> Let f be a program (function) with two parameters k (a known parameter) and u (an unknown parameter). First, finish all the f computation that can be performed by using only the k value and leave intact the f computation that cannot be performed without knowing the u value. Then a new program f_{k0} is generated having the property:

$$f_{k0}[u] = f[k0; u] \qquad (1)$$

where $k0$ stands for the k value.

Equation 1 is similar to that of Kleene's *s–m–n theorem* [Kle52] as first pointed out by [Ers78]. However, f_{k0} of the *s–m–n theorem* is just a pair of an original program and its data such as $\lambda[[u]; f[k0; u]]$. On the contrary, since the computation concerning k has been finished in f_{k0} produced by partial computation, the $f_{k0}[u0]$ may run quicker than $f[k0; u0]$ when a given u value is $u0$.

Example 1 ([Ers82]) Let $ackermann[m; n]$ be Ackermann's function:

- $ackermann_0[n] = n + 1$
- $ackermann_1[n] = n + 2$
- $ackermann_2[n] = 2n + 3$
- $ackermann_3[n] = 2^{n+3} - 3$

The practical importance of partial computation in computer science was first recognized in [LR64,Fut71, Dix71]. A method for generating a compiler from an interpreter using a partial evaluator (partial computation program) was developed in [Fut71,Fut73]. Quite a few compilers and compiler generators have been implemented based on the method [Kah82,JSS85,TF86, Tur86]. Its outline is described in Section 2. Reports on a variety of partial computation applications are listed in [SZ88].

Generalized Partial Computation is a program optimization principle based on partial computation and theorem proving. Conventional partial evaluators explicitly make use of only given parameter values to partially evaluate programs. However, GPC explicitly utilizes not only given values but also the following information:

1. Logical structure of a program to be partially evaluated.
2. Abstract data type of a programming language.

To show differences between the two partial evalua-

*Chief Researcher, Advanced Research Laboratory, Hitachi Ltd.

tors, a linear time Knuth-Morris-Pratt pattern matcher [KMP77] was generated by GPC partially evaluating a nonlinear time pattern matcher with respect to a given pattern [FN88].

This paper describes (1) interesting properties of partial computation, (2) differences between program evaluators, partial evaluators and generalized partial evaluators and (3) principles and applications of GPC.

2 INTERESTING PROPERTIES

This section describes very interesting properties of a partial computation program called a partial evaluator.

Let I be a programming language interpreter written in a universal meta language such as LISP. Then the language defined by I is called *I-language*. Let c^I, p and d be an I-language *compiler*, a *program* and *data*, respectively. Note that c^I is written in the meta language while p is written in I-language. Then the following relation holds:

$$c^I[p][d] = I[p;d] \tag{2}$$

where $c^I[p][d]$ is an abbreviation of $(c^I[p])[d]$. Note that $c^I[p]$ is an object program of p.

Let $\alpha[f;k0]$ be the result of partially computing f when $k = k0$, i.e. α is a partial evaluator; then $\alpha[f;k0] = f_{k0}$. Therefore, the following relation holds:

$$f[k;u] = \alpha[f;k][u] \tag{3}$$

Substitution of I, p and d for f, k and u, respectively, of Equation 3 produces:

$$I[p;d] = \alpha[I;p][d] \tag{4}$$

Substitution of α, I and p for f, k and u, respectively, of Equation 3 produces:

$$\alpha[I;p] = \alpha[\alpha;I][p] \tag{5}$$

Substitution of α, α and I for f, k and u, respectively, of Equation 3 produces:

$$\alpha[\alpha;I] = \alpha[\alpha;\alpha][I] \tag{6}$$

Therefore: $c^I[p][d] = I[p;d]$ by (2)
$= \alpha[I;p][d]$ by (4)
$= \alpha[\alpha;I][p][d]$ by (5)
$= \alpha[\alpha;\alpha][I][p][d]$ by (6)
This means that $\alpha[I;p]$ is an *object program*, $\alpha[\alpha;I]$ is an I-language *compiler* and $\alpha[\alpha;\alpha]$ is a *compiler generator*. Let $\omega = \alpha[\alpha;\alpha]$.

Equations (4), (5) and (6) are called the first, second and third *Futamura Projection*, respectively [Ers80]. These equations have been found by several researchers independently in the first half of 1970's including [Fut71, Fut73,BHOS76,Tur86,Ers88].

Another important α property is derived by substituting α for I of Equation 6 [Fut83]:

$$\omega = \omega[\alpha] \tag{7}$$

Equation 7 means that the compiler generator ω (note that it is $\alpha[\alpha;\alpha]$) is also an α-language compiler. Therefore, $\omega[f]$ is an object program of f:

$$\omega[f][k] = \alpha[f;k] \tag{8}$$

Equation 8 suggests that the partial computation of f with respect to k may be performed more efficiently through compiling f by ω than directly computing $\alpha[f;k]$.

3 DIFFERENCES

This section explains the difference between program evaluation, partial computation and generalized partial computation. LISP M-expression[M*62] are used to describe programs in the following discourse.

Let e be a program with two free variables k and u, a be its operating environment and *eval* be a program evaluator or interpreter. Environment a is a list of variable–value pairs (e.g. $a = ((k.k0)(u.u0))$). Then the result of evaluating e in the environment a is represented by $eval[e;a]$. Let *peval* be a (conventional) partial evaluator[Fut71]. The purpose of *peval* is to perform the computation of $eval[e;((k.k0))]$ as much as possible without knowing the u value. The result of partial computation is also represented by $peval[e;((k.k0))]$ having the property:

$$eval[e;((k.k0)(u.u0))] = eval[peval[e;((k.k0))];((u.u0))] \tag{9}$$

This equation is another form of Equation 1.

The *eval* and *peval* deal with conditional forms differently when the condition value is undefined (or unknown). This is the most obvious difference between the two. Let e be a conditional form such that

$$e = [p \rightarrow x; y]$$

meaning *if p then x else y*.

eval:

1. If $eval[p;a] = true$ then $eval[e;a] = eval[x;a]$.
2. If $eval[p;a] = false$ then $eval[e;a] = eval[y;a]$.
3. If $eval[p;a] = undefined$ then $eval[e;a] = undefined$.

peval:

1. If $eval[p;a] = true$ then $peval[e;a] = peval[x;a]$.

2. If $eval[p;a] = false$ then $peval[e;a] = peval[y;a]$.

3. If $eval[p;a] = undefined$ then $peval[e;a]$ generates the following conditional form:

$$[p/a \rightarrow peval[x;a]; peval[y;a]] \qquad (10)$$

where p/a is an expression obtained by substituting variable values of a to p.

Partial evaluator *peval* generates a new conditional form as its value when the p value is unknown, while program evaluator *eval* becomes undefined. This feature makes *peval* more powerful than *eval* with resect to its computational power. However, *peval* does not use the following important information:

> In Equation 10, p/a and $\neg(p/a)$ holds in $peval[x;a]$ and in $peval[y;a]$, respectly.

To use this information effectively, generalized partial evaluator β has a conjunction of predicates about variables as its operating environment i. Environment is $e = ((k.k0)(u.u0))$ for both *eval* and *peval*, while it is $i \equiv \{k = k0\} \wedge \{u = u0\}$ for β. Instead of using *eval* for evaluating condition p, β uses a theorem prover to prove p or $\neg p$ from environment i. In the following discourse, expression $i \vdash^* p$ will be used to show that p is provable from information i.

β:

1. If $i \vdash^* p$ then $\beta[e;i] = \beta[x;i]$.

2. If $i \vdash^* \neg p$ then $\beta[e;i] = \beta[y;i]$.

3. Otherwise, $\beta[e;i]$ generates the following conditional form:

$$[p \rightarrow \beta[x; i \wedge p]; \beta[y; i \wedge \neg p]] \qquad (11)$$

In the case 3 of β, *otherwise* may mean that neither p nor $\neg p$ is provable by a computer within a predetermined time period. More precise definition of β is given in Section 4.

Note that theorem proving and generation of a new predicate have been conducted in symbolic execution [CHT79] and program verification [NO79] as in β. However, they have never had the function of generating a conditional form described above.

Partial evaluators also deal with a recursive function call differently from program evaluators. Since partial evaluators try to evaluate an expression with an unknown value, terminating a recursive call is a difficult problem for them. This problem is discussed in Section 5.

Now, α can be defined by using β as follows:

$$\alpha[f;k0] = \lambda[[u];\beta[e;\{k = k0\}]]$$

where $f = \lambda[[k;u];e]$. This equation shows the fact that β is more general than α. That is the reason for how β is called.

4 GPC METHOD

The Generalized Partial Computation (GPC) method has been established by formalizing human informal program transformation processes (getting a fixpoint of a recursive function[Man74] is an example of such a transformation). GPC uses a logic system to evaluate a predicate which is not evaluable by an interpreter (or program evaluator). The logic system is consistent with the interpreter and is called the underlying logic. Before explaining the basic idea of GPC, definitions will be provided for a u-form, u-information and the underlying logic.

Definition 1 A *u-form* is $\perp$, a constant, variable u, or a LISP form including only u as a free variable. The symbol $\perp$ is called *bottom* and is used to stand for an undefined value.

The following is an example of a u-form:

$$[u = 0 \rightarrow 1; u * fact[u - 1]]$$

When a form includes more than one variable, for example x and y, then the variables are treated as a variable-vector such as $u =< x, y >$.

Definition 2 Let e be a u-form and *eval* be an interpreter of u-forms. Then $eval[e;((u.c))]$ stands for a *value* of e when the constant c is a value of u. If the value of $eval[e;((u.c))]$ is undefined then $eval[e;((u.c))] = \perp$.

Definition 3 Let a and b be constants or $\perp$. Then $a \preceq b$ if and only if $a = b$ or $a = \perp$.

Definition 4 *u-information* is a conjunction of predicates on u. Note that this is also a u-form.

Examples of u-information are $\neg null[u] \wedge (A = car[u]) \wedge null[cdr[u]]$ and *true*. Symbol ϕ is used to represent *true*.

Definition 5 Let i and p be any u-information such that $i \vdash^* p$, where $i \vdash^* p$ means that p is provable form i based on some logic system. The logic system is *compatible* with *eval* if and only if $eval[p;((u.c))] \preceq true$ for any constant c such that $eval[i;((u.c))] \preceq true$.

When $eval[p;((u.c))]$ is always *true* or *false* (i.e. defined) for predicate p and any constant c, then $\preceq$ is equivalent to $=$. The compatibility property guaran-

tees the soundness of a logic system with respect to the interpreter *eval*.

Example 2 Let $L0$ be a logic system in which $i = p$ if $i \vdash^* p$. Then $L0$ is a trivial logic system that is compatible with *eval*.

Definition 6 The logic system is called an *underlying logic* if and only if it is compatible with *eval*.

Depending on the predicate evaluation power of *eval*, an underlying logic can be any logic system, for example propositional logic, predicate logic, or informal logic.

Definition 7 Let L be an underlying logic, e be a u-form and i be a u-information. Then any transformation β of e to a u-form using L and i is called a *generalized partial computation method*. The result of the transformation is written as $\beta[L; e; i]$.

Definition 8 The pair β–L is called the β–L *partial evaluator* or the β–*partial evaluator* if L is not very important. When there is no confusion, β can stand for both the partial computation method and the β partial evaluator.

For example, β in $\beta[e; i]$ in the following discourse means a β–L partial evaluator, and $\beta[e; i] = \beta[L; e; i]$ for some L. When β is also clear in the context, $(e)_i$ is used to represent $\beta[e; i]$.

In the following discourse, β and L stand for a nonspecific partial computation method and its underlying logic, respectively. While, $\beta 0, \beta 1, \beta 2, \beta 3 \ldots$ stand for specific partial computation methods.

Example 3 Let $\beta 0$ be a transformation such that

$$\beta 0[e; i] = e$$

Then $\beta 0$ is a very trivial partial computation method.

Partial Computation Method $\beta 1$:

1. If e is a conditional expression such that:

$$e = [p \rightarrow x; y]$$

then

   (a) If $i \vdash^* p$, then $(e)_i = (x)_i$.

   (b) If $i \vdash^* \neg p$, then $(e)_i = (y)_i$.

   (c) If it is not easy to decide if $i \vdash^* p$ or $i \vdash^* \neg p$, then

$$(e)_i = [p \rightarrow (x)_{i \wedge p}; (y)_{i \wedge \neg p}]$$

2. If e is not a conditional expression, then $(e)_i = e$ (i.e. there is no transformation).

End of $\beta 1$.

Note that $\beta 1[L0; e; \phi] = e$.

Example 4 Assume that $a[m; n] = [m = 0 \rightarrow n + 1; g[m; n]]$ and $\beta 1$ uses informal logic on natural numbers as its underlying logic. Then

$\beta 1[a[m; n]; m = 0] = n + 1$ and

$\beta 1[a[m; n]; m > 0] = g[m; n]$

Definition 9 Let d and e be u-forms, and i be u-information. Then $d \preceq^i e$ if and only if $eval[d; ((u.c))] \preceq eval[e; ((u.c))]$ for any constant c such that $eval[i; ((u.c))] \preceq true$.

This means that e is more defined than d for a constant c when c does not make i false. Therefore, $d \preceq^\phi e$ means that the domain of e is larger than that of d.

Definition 10 (correctness of β partial evaluator) β partial evaluator is correct if and only if $e \preceq^i \beta[e; i]$ for any u-form e and any u-information i.

The correctness of the $\beta 0$ partial evaluator and β-$L0$ partial evaluator is trivial. Let f be an undefined function, and e be a u-form described below:

$$e = [f[u] = 0 \wedge f[u] \neq 0 \rightarrow 1; 0]$$

Then Definition 10 suggests that $\beta[e; i]$ may be 0 while $eval[e; ((u.c))]$ for any c is undefined. Therefore, program transformation by partial evaluators does not always preserve least fixpoints of programs.

Definition 11 Partial computation method β is correct if and only if β-L partial evaluator is correct for any underlying logic L.

It is trivial that partial computation method $\beta 0$ is correct.

Theorem 1 Partial computation method $\beta 1$ is correct.

Two lemmas are given before the proof of Theorem 1.

Lemma 1 Let i be any u-information and e be a conditional u-form such that

$$e = [p \rightarrow x; y]$$

then

1. if $i \vdash^* p$ then $e \preceq^i x$ and
2. if $i \vdash^* \neg p$ then $e \preceq^i y$.

Proof of Lemma 1: (1) If $i \vdash^* p$ then $eval[p;((u.c))] \preceq true$ for any c such that $eval[i;((u.c))] \preceq true$ from the definition of the underlying logic. Therefore, $eval[e;((u.c))]$ is $\perp$ or $eval[x;((u.c))]$. Therefore, $e \preceq^i x$. (2) The same as (1). (QED)

Lemma 2 Let e be a conditional u-form the same as above and $e1$ be a conditional u-form described below:

$$e1 = [p \rightarrow x1; y1]$$

If $x \succeq^{i \wedge p} x1$ and $y \succeq^{i \wedge \neg p} y1$ then $e \succeq^i e1$.

Proof of Lemma 2: Assume that $eval[i;((c.u))] \preceq true$. If $eval[p;((u.c))] = \perp$ then $eval[e;((u.c))] = eval[e1;((u.c))] = \perp$. If $eval[p;((u.c))] = true$ then $eval[e;((u.c))] = eval[x;((u.c))] \succeq eval[x1;((u.c))] = eval[e1;((u.c))]$. When $eval[p;((u.c))] = false$, it can be proved that $eval[e;((u.c))] \succeq eval[e1;((u.c))]$ almost the same above. Therefore, $e \succeq^i e1$. (QED)

Proof of Theorem 1: Let e be a u-form and i be u-information. By using induction on the nesting depth of conditional forms in e, $\beta 1[e;i] \succeq^i e$ will be proved.

1. When e is not a conditional form, $\beta 1[e;i] = e \succeq^i e$.
2. When e is a conditional form,
   (a) Assume that $i \vdash^* p$.
   Then $\beta 1[e;i] = \beta 1[x;i] \succeq^i x$ (from the induction hypothesis)
   $\succeq^i e$ (from Lemma 1)
   (b) Assume that $i \vdash^* \neg p$. This case is almost the same as above.
   (c) Assume that neither $i \vdash^* p$ nor $i \vdash^* \neg p$. Then $\beta 1[e;i] = [p \rightarrow \beta 1[x;i \wedge p]; \beta 1[y;i \wedge \neg p]]$ and $\beta 1[x;i \wedge p] \succeq^{i \wedge p} x$ and $\beta 1[y;i \wedge \neg p] \succeq^{i \wedge \neg p} y$ from the induction hypothesis. Therefore, from Lemma 2,

$$\beta 1[e;i] \succeq^i [p \rightarrow x; y] = e$$

   (QED)

Definition 12 Let d and e be u-forms and i be a u-information. $d \equiv^i e$ if and only if $d \preceq^i e$ and $e \preceq^i d$.

Symbol $\equiv^i$ stands for a kind of a strong equivalence. Symbol $\equiv^\phi$ stands for a strong equivalence itself. Symbol $\equiv$ will be used as an abbreviation for $\equiv^\phi$.

Definition 13 A β partial evaluator is strictly correct if and only if $e \equiv^i \beta[e;i]$ for any u-form e and any u-information i.

If β partial evaluator is *strictly correct*, then $e \equiv \beta[e;\phi]$. This means that $eval[e;((u.c))] = eval[\beta[e;\phi];((u.c))]$ for any constant c. Therefore, the transformation $\beta[e;\phi]$ by a strictly correct β partial evaluator preserves the least fixpoint of e.

Theorem 2 If every predicate, say p, in underlying logic is total, i.e. $eval[p;((u.c))]$ is defined for any constant c, then $\beta 1$ partial evaluator is strictly correct.

Proof of Theorem 2: Replace $\succeq^i$ by $\equiv^i$ in the proof of Theorem 1. (QED)

Note that u-information is a dynamic part of the information about the operating environment of a program. It varies during partial computation depending on program structure. On the contrary, abstract data type or information about functions (for example $car[cons[x;y]] = x$) does not vary during partial computation, i.e. it is static. Let $g0$ be static information, i be dynamic u-information, g be a higher order variable with its domain of predicates, and $\beta'[e;i;g]$ be $\beta[e;i \wedge g]$. Then $\beta'_{g0}[e;i] = (\beta[e;i \wedge g])_{g=g0}$. Therefore, β'_{g0} is a partial evaluator including $g0$ static information in it. Thus, generality will not be lost if it is thought that static information is included in a partial evaluator.

5 MORE PRACTICAL GPC

As described in Section 4, $\beta 0$ partial evaluator is correct for any underlying logic. Therefore, there are an infinite number of correct partial evaluators. However, $\beta 0$ partial evaluator has no practical significance because it does not improve program efficiency. $\beta 1$ is still far from being practical because it does not perform any transformation for non conditional u-forms. In this section, partial computation method $\beta 2$ that performs significant transformation on u-forms is described. $\beta 2$ changes u-form e depending on the type of e such as a constant, a variable, or a composite expression. As before, b/g stands for a u-form obtained from b, substituting g for all the free occurrences of u in b. For example, if $b = car[u]$ and $g = cdr[u]$ then $b/g = car[cdr[u]]$.

$\beta 2$ handles conditional u-forms the same as $\beta 1$. The hardest point in implementing $\beta 2$ is when e is a composite u-form. Let $e = f[g]$ where f is a function not including u as a free variable and g is a u-form. Furthermore, let $f = \lambda[[u];b]$ for a u-form b. To carry out partial computation of e with respect to u-information i in this case, just replacing $(f[g])_i$ by $(b/g)_i$ is not enough. This is because when b includes recursive calls to f, the substitution often causes infinite repetition of similar computation. A technique called *partial definition* is introduced below to eliminate the repetition as often as possible.

Before starting partial computation of u-form $f[g]$ with respect to u-information i, let f_g^i be a new function name as a result of the partial computation. f, g and i are called a nonprimitive function, a symbolic argument and partial information, respectively. f_g^i is called a partially defined function for $(f[g])_i$. After completing the partial computation, f_g^i is finally defined. However, the fact that f_g^i will be the result of partial computation may be used during the partial computation. This is a sort of indirect addressing.

A program transformation technique using f_g^i during partial computation has already been developed in [Fut71]. This is a special case of a general program transformation technique called folding[BD77]. Introducing a partially defined function is nearly equal to adding the rule $f_g^i \Leftarrow \beta 2[f[g]; i]$ to a system of recursion equations and then continuing the partial computation. The use of a partially defined function f_g^i is similar to a folding which replace $f_g^i[u]$ for $\beta 2[f[g]; i]$.

Let H be a global set of functions which is empty before starting partial computation. Using H, partial functions and partially defined functions will be defined below.

Definition 14 Let i be u-information and $e = f[g]$ be a u-form. Then $(e)_i$ is *partially defined* if and only if there is u-information j such that

$$i \vdash^* j/k \text{ and } f_d^j[u] \in H$$

where d and k are u-forms such that $g = d/k$.

Definition 15 Function f_d^j in Definition 14 is called a *partially defined function* for $(e)_i$.

Example 5 Partially defined functions for $(e)_i$ where $e = f[g]$:

1. Let $j = \phi$, i be any u-information, and d and k be any u-forms such that $g = d/k$. If $f_d^j \in H$, then f_d^j is a partially defined function for $(e)_i$ because $i \vdash^* \phi$ and $\phi/k = \phi$.
2. Let $g = cdr[u], k = cdr[u], d = u$ and $j2 = \phi$. If $f_u^\phi \in H$, then f_u^ϕ is a partially defined function of $(e)_i$ because of 1.
3. Let $g = k = cdr[u], d = u$, $j3 = \neg null[cd^2r[u]] \wedge (cadr[u] = A) \wedge (car[u] = A)$, $i = \neg null[cd^3r[u]] \wedge \neg(cad^3r[u] = B) \wedge (cad^2r[u] = A) \wedge (cadr[u] = A) \wedge (car[u] = A)$. If $f_d^{j3} \in H$, then f_d^{j3} is a partially defined function for $(e)_i$ because $i \vdash^* j3/cdr[u]$.

The two partially defined functions f_d^{j2} and f_d^{j3} in the examples above are for $(f[g])_i$. Since $j3 \vdash^* j2$, $j3$ is called to be *closer* to i than $j2$, and $j2$ is called to be *further* from i than $j3$. It is clear that ϕ is the furthest from any i.

Definition 16 (use of partially defined function) Let f_d^j be a partially defined function for $(f[g])_i$. Then, computing $(f_d^j[k])_i$ instead of computing $(f[g])_i$ is called *the use of a partially defined function.*

The use of a partially defined function f_g^i causes the introduction of recursive calls to f_g^i. Therefore, the result of partial computation is a set of recursive functions. This recursion introduction has the following two effects:

1. It may dramatically increase the effectiveness of partial computation by partially computing the partial result of f_g^i.
2. It may terminate an infinite partial computation caused by repetition of similar computation.

Note that 1 and 2 above are exclusive of each other. When effect 1 is not expected, i.e. when a program will not be improved, the result of partial computation will be too large or partial computation will not terminate. Thus, effect 2 is expected. Selecting either 1 or 2 is not decidable. However, a practical heuristic automated method for the selection is an interesting future problem. Partial definition and its proper use may be essential to implementing practical partial evaluators.

To implement partially defined functions, $\beta 2$ uses the *partial definition operator* $\Leftarrow$. Let f be a function name. Then $f[u] \Leftarrow \beta 2[e; i]$ (or $f[u] \Leftarrow (e)_i$) means that when f is referred after the execution of $\Leftarrow$ operator, the body of f is the result of transformation of e by $\beta 2[e; i]$ at the time of f reference. Therefore f is a dynamically changing nonprimitive function.

Partial computation method $\beta 2$:

1. If e is a conditional form then do the same as $\beta 1$.
2. If e is a constant then $(e)_i = e$.
3. If e is a variable then $(e)_i = e$.
4. If e is a composite form such as $e = f[g]$ for a function f,
   (a) If f is a primitive function such as LISP SUBR, then $(e)_i = f[(g)_i]$.
   (b) If f is a nonprimitive function such as LISP EXPR, then let $f = \lambda[[u]; b]$ and :
      i. If $(e)_i$ is *partially defined*, then let $f_d^j = \lambda[[u]; m]$ be one of the partially defined functions (if a function with the closest partial information j to i is selected, the partial evaluator can be executed most quickly). Let $g = d/k$ and $i \vdash^* j/k$, then $(e)_i = (f_d^j[k])_i$.

ii. If $(e)_i$ is *not partially defined*, then select one of the following operations, *cont* inue or *term*inate, depending on its effectiveness (note that this selection is up to the user of the partial evaluator):

cont If it is effective in performing further partial computation, then $(e)_i = f_g^i[u]; H = H \cup f_g^i; f_g^i[u] \Leftarrow (b/g)_i$.

term Otherwise, $(e)_i = e$.

End of $\beta 2$.

Example 6 Let $f[x]$ be:

$$f[x] = [null[x] \rightarrow a; \; cons[car[x]; f[cdr[x]]]]$$

Partial evaluation of f with respect to ϕ is:

$$\begin{aligned}
&(f[x])_\phi = f_x{}^\phi[x]; H = f_x{}^\phi \\
&\qquad \text{(from cont)} \\
&\Leftarrow ([null[x] \rightarrow a; \; cons[car[x]; f[cdr[x]]]])_\phi \\
&= [null[x] \rightarrow (a)_{null[x]}; \; (cons[car[x]; f[cdr[x]]])_{\neg null[x]}] \\
&\qquad \text{(from 1)} \\
&= [null[x] \rightarrow a; \text{ (from 2)} \\
&\qquad cons[car[x]; (f[cdr[x]])_{\neg null[x]}]] \\
&\qquad \text{(from 4.a)} \\
&= [null[x] \rightarrow a; \\
&\qquad cons[car[x]; (f_x{}^\phi[cdr[x]])_{\neg null[x]}]] \\
&\qquad \text{(from 4.b.i)} \\
&= [null[x] \rightarrow a; \\
&\qquad cons[car[x]; f_x{}^\phi[cdr[x]]]] \\
&\qquad \text{(from term)}
\end{aligned}$$

$\beta 2$ terminates when e is a constant or a variable, or when the user of $\beta 2$ decides to terminate. Finding practical methods for automatic termination is an interesting research problem.

The above example is to show how partially defined funcions are used. More practical examples have been shown in [FN88]. One of them is:

$\beta 2$[simple-pattern-matcher[pattern;text];
{pattern=given-pattern}]
=Knuth-Morris-Pratt-pattern-matcher[text]

In [FN88], partial computation method $\beta 3$ has been presented. $\beta 3$ is more powerful than $\beta 2$. One example is:

$\beta 3$[McCarthy's-91-function[n],ϕ]
$= [n > 100 \rightarrow n - 10; 91]$

6 CONCLUSION

GPC is an amalgamation of a program evaluator and a theorem prover. It has been shown that GPC is able to evaluate much more varieties of programs than program evaluators. A conventional computer is a hardware implementation of a *machine language* program evaluator. The machine accepts only a program with all parameter values. It is programmers to partially evaluate their programs to generate better ones. Examples of this paper tell that partial evaluation is not a humane task. If GPC machine were implemented, programmers could avoid doing partial evaluation by themselves and program development on the machine would be easier than before.

ACKNOWLEDGEMENTS

The author is grateful to Prof. T. E. Cheatham of Harvard University and Dr. E. Maruyama of Hitachi Advanced Research Laboratory for their encouragements and supports which enable the author to continue partial computation research.

References

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, 1977.

[BHOS76] L. Beckman, A. Haraldson, O. Oskarson, and E. Sandewall. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7(4):319–357, 1976.

[CHT79] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and analysis of programs. *IEEE Trans. on Software Engineering*, 5(4), 1979.

[Dix71] J. Dixon. The specializer, a method of automatically writing programs. Division of Computer Research and Technology, NIH, Bethesda, Maryland, 1971.

[Ers78] A. P. Ershov. Mixed computation in the class of recursive program schema. *Acta Cybernetica*, 4(1), 1978.

[Ers80] A. P. Ershov. Futamura projection. *bit*, 12(14), 1980. in Japanese.

[Ers82] A. P. Ershov. Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, (18), 1982.

[Ers88] A. P. Ershov. Opening key–note speach. *New Generation Computing*, 6(2 and 3), 1988. Special Issue on Partial Evaluation and Mixed Computation.

[FN88] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner et al., editors, *Proc. of Workshop on Partial Evaluation and Mixed Computation, Ebberup, Denmark, October, 1987*, North–Holland, 1988.

[Fut71] Y. Futamura. Partial evaluation of computation process—an approach to a compiler–compiler. *Computer, systems, controls*, 2(5):45–50, 1971.

[Fut73] Y. Futamura. El 1 partial evaluator. DEAP, Harvard University, 1973. Term paper manuscript, AM260.

[Fut83] Y. Futamura. Partial computation of programs. In E. Goto et al., editors, *Lecture Notes in Computer Science 147*, pages 1–35, RIMS Symposia on Software Science and Engineering, Kyoto, Japan, 1982, Springer–Verlag, 1983.

[JSS85] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J. -P. Jouannaud, editor, *Lecture Notes in Computer Science Vol.202*, pages 124–140, Rewriting Techniques and Applications, Springer-Verlag, 1985.

[Kah82] K. M. Kahn. A partial evaluator of lisp programs written in prolog. In M. Van Caneghem, editor, *Proc. of First International Logic Programming Conference, Marseille, France*, pages 19–25, 1982.

[Kle52] S. C. Kleene. *Introduction to Meta-Mathematics*. North–Holland, Amsterdam, 1952.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computer*, 6(2):323–350, June 1977.

[LR64] L. A. Lombardi and B. Raphael. Lisp as the language for an incremental computer. In E. Berkley and D. Bobrow, editors, *The Programming Language LISP: Its Operation and Application*, MIT Press, Cambridge, Massachusetts, 1964.

[M*62] J. McCarthy et al. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Massachusetts, 1962.

[Man74] Z. Manna. *Mathematical Theory of Computation*. McGRAW–HILL, New York, 1974.

[SZ88] P. Sestoft and A. V. Zamlin. Literature list for partial evaluation and mixed computation. In D. Bjørner et al., editors, *Proc. of Workshop on Partial Evaluation and Mixed Computation, Ebberup, Denmark, October, 1987*, North–Holland, 1988.

[TF86] A. Takeuchi and K. Furukawa. Partial evaluation of prolog programs and its application to meta programming. In H. -J. Kugler, editor, *Information Processing 86*, pages 415–420, North-Holland, 1986.

[Tur86] V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3):292–325, July 1986.