Essence of generalized partial computation

Yoshihiko Futamura
Department of Information and Computers, School of Science and Engineering, Waseda University, Ohokubo 3-4-1, Sinjuku-ku, Tokyo 169, Japan

Kenroku Nogi and Akihiko Takano
Advanced Research Laboratory, Hitachi Ltd., Hatoyama-machi, Saitama 350-03, Japan

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. The main purpose of this paper is to present comprehensible examples of GPC. Graphical notations, called GPC trees, are introduced to visibly describe GPC processes.

1. Introduction

Generalized Partial Computation (GPC) is a program optimization principle based on partial computation and theorem proving. The idea of GPC was introduced in [16, 17] where two examples (McCarthy’s 91 function and a pattern matcher) were presented to demonstrate the power of GPC. However, the explanations of the results were not very clear because the authors did not have a method to clearly describe such a complicated process as GPC. Here, we are going to give a clear description of GPC processes using new graphical notations called GPC trees. Before explaining what GPC trees are, partial computation (PC) and GPC will be reviewed briefly.

Partial computation (PC) is a systematic method of generating an efficient program based on a given program and a part of its data [22]. PC of $f$ with respect to $k_0$ is defined as follows [13]: Let $f$ be a program (function) with two parameters $k$ (known) and $u$ (unknown). 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 $f_{k_0}$ is generated having the
property

\[ f_{k0}(u) = f(k0, u) \]  (1)

where \( k0 \) stands for the \( k \) value. Since the computation concerning \( k \) has been finished in \( f_{k0} \), the \( f_{k0}(u0) \) may run quicker than \( f(k0, u0) \) when a given \( u \) value is \( u0 \).

Let \( \text{human} \) be a program with two parameters \( \text{knowledge} \) and \( \text{problem} \). Then creating a specialist \( \text{human}_{\text{knowledge}} \) from \( \text{human} \) and \( \text{knowledge} \) is a good example of partial computation:

\[ \text{human}_{\text{knowledge}}(\text{problem}) = \text{human}(\text{knowledge}, \text{problem}) . \]

Specialist \( \text{human}_{\text{knowledge}} \) can solve problems much quicker than ordinary \( \text{human} \) when the problems are covered by his specific \( \text{knowledge} \).

Another good example of partial computation is specialization of Ackermann’s function which was first discussed by Ershov [10, 11]. Let \( f \) be Ackermann’s function, i.e.

\[
\begin{align*}
  f(m, n) &= \text{if } m = 0 \text{ then } n + 1 \\
  &\quad \text{else if } n = 0 \text{ then } f(m - 1, 1) \\
  &\quad \text{else } f(m - 1, f(m, n - 1)).
\end{align*}
\]

Then \( f_0(n) = n + 1, f_1(n) = n + 2, f_2(n) = 2n + 3, f_3(n) = 2^{n+3} - 3 \) and so on.

The results of partial computation presented above are much simpler than Ershov’s which contain complicated mutual recursive functions. We will show our method to specialize Ackermann’s function in Example 10 of Section 4.

Now consider self-application of a partial evaluator. A partial evaluator \( \alpha \) is a program with two parameters \( f \) and \( k \) such that

\[
\alpha(f, k) = f_k.
\]  (2)

From (2), the following two equations can be derived:

\[
\begin{align*}
\alpha(\alpha, k) &= \alpha_k, \quad (3) \\
\alpha(\alpha, \alpha) &= \alpha_\alpha. \quad (4)
\end{align*}
\]

By using the \( \alpha \), specialist \( \text{human}_{\text{knowledge}} \) can be generated from \( \text{human} \) and \( \text{knowledge} \):

\[ \alpha(\text{human}, \text{knowledge}) = \text{human}_{\text{knowledge}} \quad \text{(by (2))}. \]

Therefore, the \( \alpha \) is considered to be the trainer of specialist. In the same way as above:

\[ \alpha(\alpha, \text{human})(\text{knowledge}) = \alpha_{\text{human}}(\text{knowledge}) = \text{human}_{\text{knowledge}} \quad \text{(by (3))}. \]

This equation means that the \( \alpha_{\text{human}} \) is the personal trainer of a specific \( \text{human} \). In the same way as above:

\[ \alpha(\alpha, \alpha)(\text{human}) = \alpha_\alpha(\text{human}) = \alpha_{\text{human}} \quad \text{(by (4))}. \]

This equation means that \( \alpha_\alpha \) is the creator of a personal trainer.
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.

This paper discusses (1) interesting properties of PC, (2) differences between conventional PC and GPC, (3) GPC trees which are graphical notations to describe GPC processes, (4) termination conditions for GPC processes.

2. Interesting properties

This section describes interesting properties of the partial evaluator $\alpha$, a part of which has been discussed in Section 1.

Let $I$ be a programming language interpreter written in a universal meta language such as LISP. Then the language defined by $I$ is called an $I$-language. Let $c'$, $p$ and $d$ be an $I$-language compiler, a program and data, respectively. Note that $c'$ is written in the meta language while $p$ is written in $I$-language. Then the following equation defines the relationship between a compiler and an interpreter [13, 14]:

$$c'(p)(d) = I(p, d).$$  \hspace{1cm} (5)

Note that $c'(p)$ is an object program (i.e. a compiled code) of $p$. By equations (1) and (2), the following relation holds:

$$f(k, u) = \alpha(f, k)(u).$$  \hspace{1cm} (6)

Substitution of $I$, $p$ and $d$ for $f$, $k$ and $u$, respectively, in (6) produces

$$I(p, d) = \alpha(i, p)(d).$$  \hspace{1cm} (7)

Substitution of $\alpha$, $I$ and $p$ for $f$, $k$ and $u$, respectively, in (6) produces

$$\alpha(I, p) = \alpha(\alpha, I)(p).$$  \hspace{1cm} (8)

Substitution of $\alpha$, $\alpha$ and $I$ for $f$, $k$ and $u$, respectively, in (6) produces

$$\alpha(\alpha, I) = \alpha(\alpha, \alpha)(I).$$  \hspace{1cm} (9)

Therefore

$$c'(p)(d) = I(p, d) \quad \text{(by (5))}$$

$$= \alpha(I, p)(d) \quad \text{(by (7))}$$

$$= \alpha(\alpha, I)(p)(d) \quad \text{(by (8))}$$

$$= \alpha(\alpha, \alpha)(I)(p)(d) \quad \text{(by (9))}.$$
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. These facts were called Futamura projections although they were discovered by several researchers including [2, 10, 13, 14, 28] independently in the early 1970s. Ershov described the history of the discovery in [12]. Quite a few compilers and compiler generators have been implemented based on the method discussed above [9, 18, 19, 20, 27, 28]. Reports on a variety of partial computation applications are listed in [25].

Another important property is derived by substituting $\alpha$ for $I$ of equation (9) and using equation (4) [15]:

$$\alpha_\alpha = \alpha_\alpha(\alpha).$$

Equation (10) means that the compiler generator $\alpha_\alpha$ is also an $\alpha$-language compiler. Therefore, $\alpha_\alpha(f)$ is an object program of $f$:

$$\alpha_\alpha(f)(k) = \alpha(f, k).$$

Equation (11) suggests that the partial computation of $f$ with respect to $k$ may be performed more efficiently through compiling $f$ by $\alpha_\alpha$ than directly computing $\alpha(f, k)$. (The application of this equation to a pattern matcher will be discussed in Example 11 of Section 4).

3. Differences

This section described differences between conventional PC and GPC. Let $e$ be a program with two free variables $k$ and $u$, $a$ be its operating environment, and $eval$ be a program evaluator. 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. 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 the partial computation is also represented by $peval(e, ((k, k0)))$ having the property

$$eval(e, ((k, k0)(u, u0))) = eval(peval(e, ((k, k0))), ((u, u0))).$$

This equation is another form of equation (1).

The $eval$ and $peval$ deal with conditional forms differently when the condition values are unknown. This is the most distinguished difference between the two evaluators. Let $e$ be a conditional form such that if $p$ then $x$ else $y$. Partial evaluator $peval$ generates a new conditional form if $peval(p, a)$ then $peval(x, a)$ else $peval(y, a)$ as its value when the $p$ value is unknown, while the program evaluator $eval$ becomes undefined. This feature makes $peval$ more computationally powerful than $eval$. However, $peval$ does not use the following important information:

Even if the $p$ value is unknown, condition $p$ holds in the $\text{then}$-part and $\neg p$ holds in the $\text{else}$-part.
To use this information effectively, the generalized partial evaluator $\beta$ has a conjunction of predicates about variables as its operating environment $i$. The environment is $a = ((k, k0)(u, u0))$ for both $eval$ and $peval$, while it is $i = (k = k0) \land (u = u0)$ for the $\beta$. Instead of using $eval$ for evaluating condition $p$, $\beta$ uses a theorem prover to prove $p$ or $\neg p$ from environment $i$. In the following, expression $\vdash i \supset p$ will be used to show that $p$ is provable from information $i$ (or $i \models p$ holds).

Using a theorem power, a generalized partial evaluator $\beta$ partially evaluates conditional form $e$ as follows:

1. if $\vdash i \supset p$ then $\beta(e, i) = \beta(x, i)$,
2. if $\vdash i \supset \neg p$ then $\beta(e, i) = \beta(y, i)$,
3. otherwise, $\beta(e, i) = \text{if } p \text{ then } \beta(x, i \land p) \text{ else } \beta(y, i \land \neg p)$.

In case (3) above, otherwise may mean that neither $p$ nor $\neg p$ is provable by a computer within a predetermined time period. More precise descriptions of GPC are given in Section 4.

Note that theorem proving and generation of a new predicate have been conducted in symbolic execution [5] and program verification [24] as in the $\beta$. However, they have never had the function of generating a conditional form described above. The readers of this paper will notice the difference clearly after looking at the examples in Section 4.

Partial evaluators deal with a recursive call differently from program evaluators. Since partial evaluators try to evaluate expressions with unknown values, terminating recursive calls has been a difficult problem for them. However, the principle of Termination-on-the-Second-Call (TSC) [26] works reasonably well here as discussed in Section 5.

Now, the partial evaluator $\alpha$ in Section 1 can be defined using $\beta$ as follows;

$$\alpha(f, k0) = \lambda u. \beta(e, (k = k0))$$

where $f = \lambda ku. e$.

We will write $(e(u))_{j(u)}$ for the abbreviation of $\beta(e, j(u))$ in the following.

### 4. GPC trees

This section describes graphical notations called GPC trees to explain how GPC processes are performed. For simplicity, a program to be partially computed is such a non-primitive function as

$$f(x) = \text{if } p(x) \text{ then } b(x, f(c(x)), f(d(x))) \text{ else } a(x)$$

having the properties described below:

1. $a, c$ and $d$ do not contain a free $f$,
2. $x$ may be a list of variables $[x_1, \ldots, x_n]$,
3. $b$ is strict, i.e. call-by-value evaluation will compute the least fixpoint of the program $f$. 
The property (3) indicates that the distribution of function \( b \) over conditionals preserves meanings of programs [23], e.g.,

\[
\begin{align*}
  f(x) = & \text{if } p(x) \text{ then if } p(c(x)) \\
  & \text{then } b(x, b(c(x), f(c^2(x)), f(d(c(x)))), f(d(x))) \\
  & \text{else } b(x, a(c(x)), f(d(x)))) \\
  & \text{else } a(x).
\end{align*}
\]

The restrictions imposed on the programs above can be generalized so long as the call-by-value semantics is assumed [6].

Let \( e(u) \) be an expression consisting of a free variable \( u \), bound variables, constants and functions (both primitive and non-primitive). Let \( j(u) \) be information about \( u \) (or \( u \)-information), i.e. \( j(u) \) is a predicate on \( u \). Only the variable \( u \) can be a free variable of \( e(u) \) and \( j(u) \). Therefore, they are called \( u \)-forms.

Consider now tree diagrams called the GPC tree of \( e(u) \) with respect to \( j(u) \) (the GPC tree of \( \beta(e(u), j(u)) \) or \( (e(u)), (j(u)) \)). An example of the tree is given in Fig. 1 (in the figure, there is a program \( N1(u) \) called a corresponding program which will be explained later). The syntax of the tree is defined by a root, nodes, leaves and branches (see Fig. 2). Note that a root and leaves are special cases of nodes. Let \( N \) be a node name. Then \( I(N) \), \( E(N) \), \( B(N) \) and \( \neg B(N) \) are defined as follows:

- \( E(N) \): an expression contained in node \( N \);
- \( I(N) \): an integer expression;
- \( f(u) \):
  - \( N1(u) = \text{if } u > 70 \text{ then } 71 \text{ else } N2(u) \);
  - \( N2(u) = \text{if } u < 69 \text{ then } 71 \text{ else } f(N2(u+1)) \) ;
  - \( f(71) \);

**Fig. 1.** GPC tree of \( f(u) = \text{if } u > 70 \text{ then } u \text{ else } f(f(u+1)) \) with respect to \( \text{integer}(u) \).

\( N1(u) = \text{if } u > 70 \text{ then } 71 \text{ else } N2(u) \).
\( N2(u) = \text{if } u < 69 \text{ then } 71 \text{ else } f(N2(u+1)) \).
\( f(71) \).

\( \neg B(N) \): a tree diagram of \( \neg \beta(N) \) which will be explained later.

\( N1(u) = \text{if } u > 70 \text{ then } 71 \text{ else } N2(u) \).
\( N2(u) = \text{if } u < 69 \text{ then } 71 \text{ else } f(N2(u+1)) \).
\( f(71) \).

\( \neg B(N) \): a tree diagram of \( \neg \beta(N) \) which will be explained later.

\( N1(u) = \text{if } u > 70 \text{ then } 71 \text{ else } N2(u) \).
\( N2(u) = \text{if } u < 69 \text{ then } 71 \text{ else } f(N2(u+1)) \).
\( f(71) \).
Fig. 2. Root, node and leaf. Here, $N$ is a node name. $i(u)$, $j(u)$, $q(u)$ and $\neg q(u)$ are $u$-information which are called branch names. $\neg q(u)$ stands for the negation of $q(u)$. Note that the node name of a leaf can be an expression contained in the node.

$B(N)$: the name of a branch from node $N$;
$\neg B(N)$: the name of the other branch from node $N$ (if any);
$I(N)$: information about node $N$, i.e. conjunction of all branch names which appear on the path from the root of the tree to node $N$ (see Example 1).

Since node names are used as function names later, $I(N)$ stands for a predicate describing the domain of $N$. The function $I$ is extended to be defined on any non-primitive function $f$.

**Example 1.** Let $N_1$, $N_2$ and $N_3$ be node names in Fig. 1:

$E(N_1) = f(u)$,
$E(N_2) = f(f(u + 1))$ or $f(N_1(u + 1))$,
$E(N_3) = f(u + 1), f(71)$ or 71,
$B(N_1) = u > 70$, $\neg B(N_1) = u \leq 70$,
$B(N_2) = u > 69$, $\neg B(N_1) = u \leq 69$,
$B(N_3) = u > 70$,
$I(N_1) = integer(u)$,
$I(N_2) = integer(u) \land u \leq 70$,
$I(N_3) = integer(u) \land u \leq 70 \land u > 69$,
$I(f) = integer(u)$.

In the following, $k(u)$ stands for an expression consisting of a free variable $u$, bound variables, constants and primitive functions (i.e. $k(u)$ does not contain neither conditionals, node names nor non-primitive functions). The $k(u)$ is called primitive $u$-form.

Following [1], substitution and context are defined below.
Definition 1 (Substitution). Let g be an expression. Then \( g[u := k(u)] \) is an expression obtained by replacing all occurrences of u in g by k(u).

Example 2. Consider node N1 in Fig. 1. Then \( E(N1)[u := u + 1] = f(u + 1) \).

Definition 2 (Context). A context C[ ] is an expression with some holes in it.

Example 3. \( C[ ] = f(\[ \]) \) is a context. \( C[f(u + 1)] = f(f(u + 1)), C[N1(u + 1)] = f(N1(u + 1)) \) etc. (The meaning of \( A = B \) is that A and B are equal as strings).

Now we will define concepts concerning generation rules of GPC trees such as P-redex, unfolding, folding, simplification and distribution.

Definition 3 (P-redex). Let h be a non-primitive function or a node name in node N, and u be free in N. Then \( h(k(u)) \) in N is the P-redex (partial computation redex) of N if and only if one of the following conditions holds:

1. if h is a non-primitive function, then \( k(Ext(I(N))) \subseteq Ext(I(h)) \) where \( Ext(i(u)) = \{u \mid i(u)\} \).
2. if h is a node name, then \( k(Ext(I(N))) \) is a proper subset of \( Ext(I(h)) \).

Since P-redex will be unfolded, the condition above guarantees that k(u) is in the domain of h. The inequality of the two sets in (2) is to avoid infinite unfolding as discussed in Section 5.

Example 4. Let N2 be a node in Fig. 1. Then \( k(Ext(I(N2))) = \{u \mid u \leq 70\} \) is a proper subset of \( Ext(I(N1)) = \{u \mid integer(u)\} \) where k(u) = u + 1. Therefore \( N1(u + 1) \) in N2 is a P-redex of N2.

Example 5. Let N4 be a leaf in Fig. 1. Then \( k(Ext(I(N4))) = Ext(I(N2)) = \{u \mid u \leq 70\} \) where k(u) = u + 1. Therefore \( N2(u + 1) \) is not a P-redex of N4. This means that node N4 has no P-redex at all.

In the next definition, we will use symbol \( \vdash \) to express that an assertion holds, e.g. \( \vdash A \implies B \) means that A implies B holds.

Definition 4 (Unfolding). Let \( H(k(u)) \) be a P-redex of leaf N and \( E(N) = C[H(k(u))] \). Then the P-redex can be transformed to one of the three structures in Fig. 3 depending upon the relationship between I(N), B(H) and \( \neg B(H) \). This transformation is called unfolding.

Example 6. Let \( N1(u + 1) \) of node N2 in Fig. 1 be \( H(k(u)) \) and \( C[\ ] = f(\[ \]) \). Then \( E(N2) = C[H(k(u))] \) and \( H(k(u)) \) is P-redex of N2 from Example 4. In addition, neither \( \vdash (u \leq 70) \supset (u > 69) \) nor \( \vdash (u \leq 70) \supset (u \geq 69) \) holds. Therefore, nodes N3 and N4 in Fig. 1 are obtained by unfolding of node N2.
Let \( H(u) = \) if \( p(u) \) then \( A(u) \) or else \( B(u) \)

\[
\begin{align*}
J(u) & \quad C[H(k(u))] \\
C[H(k(u))] & \quad \Rightarrow (1), (2) \text{ or } (3) \text{ where } \\
(1) & \quad \vdash I(N) \supset p(k(u)) \\
(2) & \quad \vdash I(N) \supset \neg p(k(u)) \\
(3) & \quad \text{neither } \vdash I(N) \supset p(k(u)) \text{ nor } \vdash I(N) \supset \neg p(k(u))
\end{align*}
\]

Definition 5 (Folding). Let \( e(u) \) be an expression in leaf \( H \) such that \( e(u) = E(N)[u := k(u)] \) and \( k(Ext(I(H))) \subseteq Ext(I(N)) \) for some node \( N \). Then leaf \( H \) is transformed to a new structure shown in Fig. 4. This transformation is called folding.

The condition \( k(Ext(I(H))) \subseteq Ext(I(N)) \) guarantees that \( k(u) \) is in the domain of \( N \).

Fig. 4. Folding. \( e(u) = E(N)[u := k(u)] \) and \( k(Ext(I(H))) \subseteq Ext(I(N)) \).
Example 7. Let $H$ and $N$ be (upper) $N_2$ and $N_1$, respectively, in Fig. 1. Let $e(u)$ be $f(u+1)$ in $N_2$ and $k(u)$ be $u+1$. Then $e(u) = E(N_1)[u := u+1]$ and $k(Ext(I(N_2))) = \{u | u \leq 71\} \subseteq \{u | integer(u)\} = Ext(I(N_1))$.

An expression in a node can be simplified anytime.

Definition 6 (Simplification). Let $e(u)$ be an expression and $e'(u)$ be a simplified expression of $e(u)$. This relationship is expressed by a single branch in Fig. 5.

Example 8. Consider node $N_3$ in Fig. 1. Since $I(N_3) = (u \leq 70) \land (u > 69)$, then $u = 70$. Therefore, $f(u+1) = f(71) = 71$.

Definition 7 (Distribution). Let $C[ ]$ be a context, $N$ be a simplified leaf such that $E(N) = C[if \ p \ then \ A \ else \ B]$, and let $p$, $A$ and $B$ be $u$-forms. Then $N$ is transformed to a structure shown in Fig. 6 (note that neither $I(N) \supset p$ nor $I(N) \supset \neg p$ holds because $N$ has been simplified).

Transformation rules such as simplification (S), distribution (D), folding (F) and unfolding (U) are called SDFU rules. The generation rule of the GPC tree of $e(u)$ with respect to $j(u)$ is defined as follows:

1. Let $N_1$ be a root such that $E(N_1) = e(u)$ and $I(N_1) = j(u)$. Note that $N_1$ is a leaf at the starting point.

Fig. 6. Distribution. The (2) is an abbreviation for (1).
To every leaf of the tree, apply simplification, distribution, folding and unfolding in that order, i.e. the SDFU order.

The generation process will terminate when there is no leaf in the tree to which SDFU rules are applicable. On termination, it is clear that the tree has no P-redex in its leaves. Termination problems will be discussed in the next section.

Now we can define the corresponding program (or function) of a GPC tree. Let $N$ be a node in a GPC tree. Then the corresponding program of $N$, i.e. $N(u)$ is:

1. If $N$ is a leaf then $N(u) = E(N)$.
2. If $N$ is a non-leaf node then $N(u)$ is shown in Fig. 7.

The corresponding program of a GPC tree is the corresponding program of its root. We will remove unnecessary node references, explicit folding and simple recursion from corresponding programs in the following. Recursion removal techniques play an important role in Examples 9 and 10.

Example 9. In Figs. 1 and 8, GPC trees and corresponding programs for double recursive functions are shown. McCarthy’s 91-function can be transformed to $f(u) = \begin{cases} u - 10 & \text{if } u > 100 \\ 91 & \text{else} \end{cases}$ in the same way as the 71-function in Fig. 1 [16].

Example 10 (Ackermann’s function). Let $f$ be Ackermann’s function, i.e.

$$f([m, n]) = \begin{cases} n + 1 & \text{if } m = 0 \\ f([m - 1, n - 1]) & \text{else if } n > 0 \\ f([m - 1, f([m, n - 1])]) & \text{else} \end{cases}.$$

The domain of $f$ is $\{[m, n] | 0 \leq m, 0 \leq n\}$. It is trivial that $f([0, u]) = u + 1$. Let $f_0(u)$ be $f([0, u])(= u + 1)$. Then Fig. 9 shows that $(f([1, u]))_{0 \leq u} = N1(u) = f_{u+1}^0(1) = u + 2$. In the same way as in Fig. 9, Fig. 10 shows the GPC tree of $(f([m + 1, u]))_{0 \leq u}$ when $(f([m, u]))_{0 \leq u} = f_m(u)$ is known for $0 \leq m$. Thus $f_m+1(u) = f_{m+1}^u(1)$. Therefore $f_2(u) = 2u + 3$, $f_3(u) = 2^{u+3} - 3$ and so on.
Fig. 8. GPC tree of $f(u) = \text{if } u \leq 0 \text{ then } u \text{ else } f(f(u-1)-1)$ with respect to $\text{integer}(u)$. $N1(u) = -|u|$ is the corresponding program of the tree.

Fig. 9. GPC tree of $f([1,u])$ with respect to $0 \leq u$ where $f([m,n])$ is Ackermann’s function and $f_0(u) = f([0,u]) = u + 1$. $N1(u)$ is the corresponding program of the tree.
Example 11 (Pattern matcher). Let $f$ be a simple pattern matcher, i.e.

$$f([p, t, w]) = \begin{cases} \text{true} & \text{if } p = [] \\ \text{false} & \text{else if } t = [] \\ \text{false} & \text{else if } \text{car}(p) \neq \text{car}(t) \\ \text{false} & \text{else if } w = [] \\ \text{false} & \text{else if } \text{car}(p) = \text{car}(t) \\ \text{false} & \text{else if } \text{car}(p) = \text{car}(t) \\ \text{false} & \text{else if } \text{car}(p) = \text{car}(t) \\ \text{false} & \text{else if } \text{car}(p) = \text{car}(t) \end{cases}$$

where $[] = \text{NIL}$ and $x :: y = \text{append}(x, y)$. Variables $p$ and $t$ stand for pattern and text. The domain of $f$ is $\{[p, t, w] \mid \text{listp}(p), \text{listp}(t), \text{listp}(w)\}$ where $\text{listp}(x)$ is a predicate to check if $x$ is a list. The GPC tree of $(f([[A, A, B], u, []])_{\text{listp}(u)})$ is shown in Fig. 11. Note that $N_1(u)$ is a linear time pattern matcher like [21]. Note also that every leaf in Fig. 11 does not contain P-redex in it. In the same way as above, we have proved that a BM-like linear pattern matcher $[-]$ can be derived from another simple pattern matcher. However, the GPC tree for the BM pattern matcher is too lengthy to be included in this paper (it is twice as large as Fig. 11). If there were such a self-applicable partial evaluator $\alpha$ described in equation (11) of Section 2, then $\alpha_\alpha(f)$ could have produced a linear pattern matcher $\alpha_\alpha(f)(p)$ much quicker (hopefully in linear time) than the method shown in Fig. 11. Actually, $\alpha(f, p)$ has been executed in Fig. 11. Although the pattern matcher example has been discussed by a few researchers [7, 8, 16], this is the first example to demonstrate the complete process of partial evaluation (although a pattern matcher example was
Fig. 11. GPC tree of \( f([p, u, [\ ]]) \) with respect to \( \text{listp}(u) \) where \( f \) is a simple pattern matcher, \( p = [A, A, B] \) and \( pn = \text{cad}^2 r(p) \). \( N1(u) \) is the corresponding program of the tree.
deal with in [3], it was not discussed in the PC context). This fact indicates that GPC trees are compact and readable enough to be used as notations for GPC process description.

5. Termination of GPC

This section discusses termination problems concerning GPC. Since the GPC tree generation process includes simplification of functions, the halting problem of the process is obviously undecidable. Problems we are going to discuss here are about termination of applying unfolding. When there is no P-redex in a GPC tree, application of unfolding terminates. However, we have not answered two questions:

1. When there is no P-redex in a GPC tree, is the corresponding program of the tree reasonably optimized? Can we get a more optimized program if we weaken the definition of P-redex so that we can continue unfolding further?

2. Is there any GPC tree which has a P-redex forever?

Before answering these questions, the Termination-on-the-Second-Call (TSC) principle is explained. This is a termination condition for partial computation first used in [13] and generalized for GPC in [26]:

If the current recursive call is in the GPC process of an equivalent function call, terminate unfolding.

The principle is embedded in the definition of P-redex (Definition 3(2)). Although the principle is not decidable, it works well as we have observed in the examples of Section 4.

5.1. Meanings of P-redex

In the previous section, all example GPC trees have no P-redex in their final shapes. In addition, their corresponding programs are reasonably optimized. However, having no P-redex is not enough for a GPC tree to claim that the corresponding program is reasonably optimized. This fact is shown by the following example.

Example 12. Let \( f \) be as follows and the domain of \( f \) be \( u > 0 \):

\[
f(u) = \text{if } u = 1 \text{ then } 1 \text{ else if } \text{odd}(u) \text{ then } f(u - 1) \text{ else } f(u/2).
\]

Then the GPC tree of \( (f(u))_{u>0} \) is shown in Fig. 12. Since if \( \text{odd}(u) \) then \( f(u - 1) \) else \( f(u/2) \) is equal to \( f((u - 1) \ast \text{mod}(u, 2) + (u/2) \ast (1 - \text{mod}(u, 2))) \), \( f(u) \) can be defined in two ways as \( f_1(u) \) and \( f_2(u) \):

\[
f_1(u) = \text{if } u = 1 \text{ then } 1 \text{ else if } \text{odd}(u) \text{ then } u - 1 \text{ else } u/2),
\]

\[
f_2(u) = \text{if } u = 1 \text{ then } 1 \text{ else } f_2((u - 1) \ast \text{mod}(u, 2) + (u/2) \ast (1 - \text{mod}(u, 2))),
\]

where \( \text{mod}(u, 2) \) is the remainder of \( u \ast 2 \).

Because of the distribution rule, the GPC tree of \( f_1(u) \) is the same as Fig. 12. Since \( (u - 1) \ast \text{mod}(u, 2) + (u/2) \ast (1 - \text{mod}(u, 2)) \) can be considered as a primitive \( u \)-form \( k(u) \), the GPC tree of \( f_2(u) \) in Fig. 13 is different from Fig. 12. The
Fig. 12. GPC tree of $f(u) = \begin{cases} 1, & u = 1 \\ \text{if } u \text{ is odd then } f(u - 1) \text{ else } f(u/2), & u > 1 \end{cases}$ with respect to $u > 0$. $N_1(u) = 1$ is the corresponding program of the tree.

Fig. 13. GPC tree of $f_2(u) = \begin{cases} 1, & u = 1 \\ f_2((u-1) \mod 2 + (u/2) \cdot (1-\mod(u,1))) & u > 1 \end{cases}$ with respect to $u > 0$. Note that no leaf has a P-redex in it.
corresponding program of the tree is the same as \( f_2(u) \) itself. Although no leaves in Fig. 13 have P-redex, no improvement has been gained by partial computation in this case.

The example above shows that not only functions but the styles of programs influence the efficiency of the corresponding programs.

**Example 13.** Let \( f(u) = \begin{cases} 1 & \text{if } u \leq 1 \\ f(u-1) + f(u-2) & \text{else} \end{cases} \). The GPC tree of \( (f(u))_{u \leq u} \) is shown in Fig. 14. Note that all the leaves in the tree have no P-redex. Although the corresponding program is more efficient than the original \( f(u) \), it still runs in exponential time like \( f(u) \). If we had extended the definition of P-redex in node \( N \) to include \( h(k(u)) \) for such a node \( h \) as \( k(\text{Ext}(I(N))) = \text{Ext}(I(h)) \), we could have produced an infinite program

\[
N1(u) = \begin{cases} \text{if } u \leq 1 \text{ then } 1 \\ \text{else if } u \leq 2 \text{ then } 2 \\ \text{else if } u \leq 3 \text{ then } 3 \\ \text{else if } u \leq 4 \text{ then } 5 \\ \text{else if } u \leq 5 \text{ then } 8 \\ \text{else} \ldots \end{cases}
\]

that runs in linear time. The example above shows that our definition of P-redex is too strong in some sense. However, if we allow \( h(k(u)) \) to be a P-redex for such a node \( h \) as \( k(\text{Ext}(I(N))) = \text{Ext}(I(h)) \), then by unfolding, \( h(k^n(u)) \) for \( n > 1 \) is a

\[
\begin{align*}
0 \leq u \\
\text{if } u \leq 1 \text{ then } 1 \\
\text{else if } u \leq 2 \text{ then } 2 \\
\text{else if } u \leq 3 \text{ then } 3 \\
\text{else if } u \leq 4 \text{ then } 5 \\
\text{else if } u \leq 5 \text{ then } 8 \\
\text{else} \ldots
\end{align*}
\]

Fig. 14. GPC tree of \( f(u) = \begin{cases} 1 & \text{if } u \leq 1 \\ f(u-1) + f(u-2) & \text{else} \end{cases} \) with respect to \( 0 \leq u \). \( N1(u) \) is the corresponding program of the tree. The \( * \) is used to specify the current P-redex for unfolding.
P-redex in the GPC tree. In this case, \( h(k(u)) \) is equivalent to \( h(u) \) in some sense [26]. Therefore, there remains a P-redex forever in the GPC tree. Based on the TSC principle, our P-redex definition avoids this infinite unfolding.

5.2. Another infinite unfolding

Let \( f \) be as follows and the domain of \( f \) be \( \{[m, n] | 0 \leq m, 0 < n \} \):

\[
f([m, n]) = \begin{cases} 
0 & \text{if } \text{mod}(m, n) = 0 \\
1 & \text{if } \text{mod}(m, n) = x \text{ else } 1.
\end{cases}
\]

Then the GPC tree of \( f([u, 2])_{0 \leq u} \) has P-redex \( f([u/n!, n + 1]) \) (where \( 2 \leq n \)) in its leaf forever. Since the TSC principle does not apply to this case, we have to stop unfolding when we know that there is this kind of an infinite process. However, we have not found a smart method for the termination yet.

6. Conclusion

GPC trees are introduced to clearly describe GPC processes. Several example processes were demonstrated, including Ackermann's function, a pattern matcher and the Fibonacci function. The authors believe that this paper is the first to give complete descriptions of PC processes for the examples. Complete descriptions of PC processes for common examples help world PC schools understand each other.

References

[14] Y. Futamura, EL1 partial evaluator, Term paper manuscript, AM260, DEAP, Harvard University, 1974.


