WSDFU: Program Transformation System Based on Generalized Partial Computation

Yoshihiko Futamura[1], Zenjiro Konishi[1], and Robert Glück[2]

[1] Waseda University, 3-4-1 Okubo, Shinjuku, Tokyo 169-8555 Japan
{futamura,konishi}@futamura.info.waseda.ac.jp
[2] PRESTO, JST & Waseda University,
3-4-1 Okubo, Shinjuku, Tokyo 169-8555 Japan
glueck@acm.org

Abstract. Generalized Partial Computation (GPC) is a program transformation method utilizing partial information about input data and auxiliary functions as well as the logical structure of a source program. GPC uses both an inference engine such as a theorem prover and a classical partial evaluator to optimize programs. Therefore, GPC is more powerful than classical partial evaluators but harder to implement and control. We have implemented an experimental GPC system called WSDFU (Waseda Simplify-Distribute-Fold-Unfold). This paper demonstrates the power of the program transformation system as well as its theorem prover and discusses some future works.

1 Introduction

Generalized Partial Computation (GPC) is a program transformation method utilizing partial information about input data, abstract data types of auxiliary functions and the logical structure of a source program. The basic idea of GPC was reported at PEMC'87 [6]. GPC uses a theorem prover to solve branching conditions including unknown variables. It also uses a theorem prover to decide termination conditions of unfolding and conditions for correct folding. The prover uses domain information about variables, abstract data types of auxiliary functions (knowledge database) and logical structures of programs to prove properties about variables and subexpressions.

Differences between classical partial evaluation [5,10,17] and GPC were described in [7]. In [7], *GPC trees* were used to describe the algorithm of GPC in detail. Although GPC was patented both in the US [8] and in Japan, it has not been fully implemented. A key element of GPC is the implementation of an efficient theorem prover.

We have tuned up Chang and Lee's classical TPU [4] for our experimental GPC system called WSDFU (Waseda Simplify-Distribute-Fold-Unfold). This paper reports program transformation methods in WSDFU, the structures of the theorem prover, benchmark tests and future works. The kernel of the system has been implemented in Common Lisp and its size is about 1500 lines of pretty-printed source code (60KB). This paper is a revision of the main part of our paper [12].

T.Æ. Mogensen et al. (Eds.): The Essence of Computation, LNCS 2566, pp. 358–378, 2002.

2 Outline of WSDFU

At present, WSDFU can optimize strict functional programs with call-by-value semantics. It can perform automatic recursion introduction and removal [2] using the program transformation system formulated by Burstall and Darlington [3,21]. We also use an algebraic manipulation system [16] to simplify mathematical expressions and to infer the number of recursive calls in a program. GPC controls all environments including knowledge database, theorem prover, recursion removal and algebraic manipulation system (Fig. 7). Program transformation in WSDFU can be summarized as follows. It proceeds in four steps:

1. Simplify expressions (programs) as much as possible (S).
2. If there is a conditional expression in a program, then distribute the context function over the conditional (D).
3. Try to fold an expression to an adequate function call (F).
4. If there is a recursive call which does not satisfy the termination conditions (W-redex), then unfold the call (U).

WSDFU iterates SDFU operations in this order until there remains no W-redex in a program. The system maintains properties of variables in the form of predicates such as $integer(u) \wedge u > 0$ or $x = 71$. Therefore, WSDFU can do all the partial evaluation so called online partial evaluators can do and more. The set of the predicates is called *environment*. The environment is modified by WSDFU in processes S and D using a theorem prover. Details of the S, D, F and U operations are described below using GPC trees. A *source program* is a program to be optimized by GPC (using WSDFU, here). Every source program becomes a node of a GPC tree with a new attached name to its node. Moreover, the GPC tree represents the *residual program* (i.e. the result of the GPC) of the source program. We now describe each process in more detail.

2.1 Simplification

Simplify a program (expression) using a theorem prover, algebraic manipulation [15,16,20] and recursion removal system [9,21]. Here, we use a knowledge database including mathematical formulas and abstract data types of auxiliary functions. Current WSDFU system has about 40 rules to solve all the examples described in this report. A simplification process is successful if the result entails the elimination of all recursive calls (to the user defined source program) through folding. Since the folding process comes after the simplification process, this operation is non-deterministic. That is, if a folding process is unsuccessful, we backtrack our SDFU process to the simplification process that caused the folding and undo the simplification. Then try another simplification if possible. Examples of simplification are shown below:

1. Let the current environment be i and a source program be a conditional expression such as **if** $p(u)$ **then** e_1 **else** e_2. If $p(u)$ is provable from i then the residual program is the result of GPC of e_1 with repect to $i \wedge p(u)$. If

$\neg p(u)$ is provable from i then the residual program is the result of GPC of e_2 with repect to $i \wedge \neg p(u)$. If neither of the above cases holds within a predetermined time, then the residual program is **if** $p(u)$ **then** (residual progam of GPC of e_1 with repect to $i \wedge p(u)$) **else** (residual program of GPC of e_2 with repect to $i \wedge \neg p(u)$).

2. Let the current environment be i and a source program be a conditional expression such as **if0** $p(u)$ **then** e_1 **else** e_2. The meaning of **if0** is the same as **if** in (1) total evaluation or (2) partial evaluation and $p(u)$ is provable or refutable. However, when $p(u)$ is neither provable nor refutable, the residual program is *residual program of GPC of* e_2 *with respect to* i. Similar to **if0**, we use **if1** the residual program of which is *residual program of GPC of* e_1 *with respect to* i. Use of **if0** or **if1** changes the semantics of residual programs and the correctness of the program transformation cannot be guaranteed. However, some times it is very useful for getting efficient residual programs. An example usage of these expressions is shown in [12].
3. Let $mod(x, d)$ be a function computing the remainder of x/d. Then $mod(x * y, d)$ is replaced by $mod(mod(x, d) * mod(y, d), d)$. The purpose of this replacement is to move the *mod* function into the multiplication. This makes the folding of composite function including *mod* easier (See Example 3.5).
4. If we know, for example, from our database or the properties of $p(x)$ and $d(x)$ that function f terminates and a is a constant, then
   $f(x) \equiv$ **if** $p(x)$ **then** a **else** $f(d(x))$
   is replaced by a.
5. If $f(a) = a$, then $f^m(a)$ is replaced by a for $m > 0$.

2.2 Distribution

Assume that a source program e contains a conditional expression such as **if** $p(u)$ **then** e_1 **else** e_2. Let e be $Cont$[**if** $p(u)$ **then** e_1 **else** e_2] for a function $Cont$. Then the operation to produce a program **if** $p(u)$ **then** $Cont[e_1]$ **else** $Cont[e_2]$ from e is called *distribution*. Since WSDFU deals with only strict programs, the distribution operation preserves semantics of programs. GPC of e wrt i produces a GPC tree shown in Fig. 1. In the figure, N_1, N_2 and N_3 are new names attached to nodes. They also stand for function names defined by GPC subtrees. For example, $N_1(u) \equiv$ **if** $p(u)$ **then** $N_2(u)$ **else** $N_3(u)$, $N_2(u) = Cont[e_1]$ and $N_3(u) = Cont[e_2]$. Environments of nodes N_2 and N_3 are $i \wedge p(u)$ and $i \wedge \neg p(u)$, respectively.

Assume that a source program e does not contain a conditional expression. Let e be $Cont[e_1]$ and e_1' be the result of GPC of e_1 wrt i. Then GPC of e wrt i produces a GPC tree shown in Fig. 2. In the figure, the environment of node N_2 is i. This process can be considered as a part of the distribution.

2.3 Folding

First, in a program e, search for a sub-expression $g(k(u))$ that has an ancestor in the current GPC tree whose body is $g(u)$ and the range of $k(u)$ is a subset of

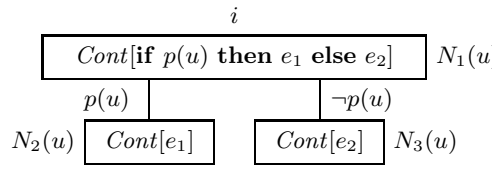

Fig. 1. GPC tree for distribution of a functional context over a conditional expression

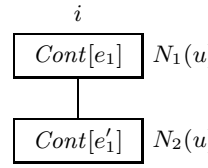

Fig. 2. GPC tree for distribution of a functional context over a simplified expression

the domain of u (Here, k is a primitive function. See Fig. 3). A theorem prover is used to check the inclusion which guarantees the correctness of our folding operation. Assume that the node name of $g(u)$ is N. Then $g(k(u))$ is replaced by $N(k(u))$.

If we find more than one $g(k(u))$'s in a predetermined time, we fold each of them one by one independently. Therefore, this process may produce more than one residual program. For example, we can choose one $g(k(u))$ which ends up with the shortest residual program. Since the shortest program is not necessarily the most efficient, users can choose the most efficient program from the residual programs if they want to have this fine level of control; otherwise a solution can be chosen automatically. We think the folding process is the most important operation in program transformation to obtain efficient residual programs [2].

2.4 Unfolding

In [7], we defined a *P-redex* as a procedure call with an actual parameter whose range is in the proper subset of the domain of the function. Among P-redexes, those who do not satisfy the termination condition (iii) or (iv) described later are called *W-redexes* (Waseda-redexes). Only W-redexes are unfolded in WSDFU. If there exists more than one W-redex in a program, we choose the leftmost-innermost W-redex first. This process can be conducted non-deterministically or in parallel like folding. Folding and unfolding are repeatedly applicable to parts of the GPC tree being processed by WSDFU.

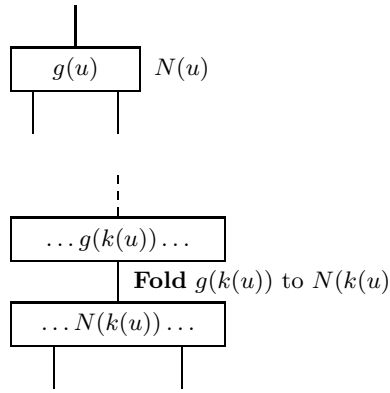

Fig. 3. Folding in a GPC tree

2.5 Example of GPC: 71-function

Here, we show the GPC of the 71-function [7] $f(u)$ wrt integer u where

$f(u) \equiv$ **if** $u > 70$ **then** u **else** $f(f(u+1))$.

Its GPC tree is shown in Fig. 4. Each N_i in the figure is a label attached to each node. The N_i stands for a function name defined by the GPC tree that has N_i as a root. WSDFU terminates at leaves N_5 and N_7 because there is no recursive call at the nodes. WSDFU also terminates at node N_4 because the termination condition (i) described later applies. Since the range of $u+1$ in N_2 is integer, underlined f can be folded to N_1. Note here that WSDFU unfold N_1 at N_3. Each edge of the tree has a predicate as a label. The conjunction of all predicates that appear on the path from the root to a node stands for the environment of variable u in the node. The residual program defined by the GPC tree is:

$N_1(u) \equiv$ **if** $u > 70$ **then** u **else** $N_3(u)$,
$N_3(u) \equiv$ **if** $u > 69$ **then** 71 **else**$f(N_3(u+1))$.

We eliminate recursion from N_3 utilizing our recursion removal system [19] and get a new node: $N_3(u) = f^{70-u}(71) = 71$. Then by simplification, N_1 is transformed to

$N_1(u) \equiv$ **if** $u > 70$ **then** u **else** 71.

This is the final residual program for $f(u)$.

3 Termination Conditions

Since, in general the halting problem of computation is undecidable, termination conditions for WSDFU have to be heuristic. Here, we discuss halting problems

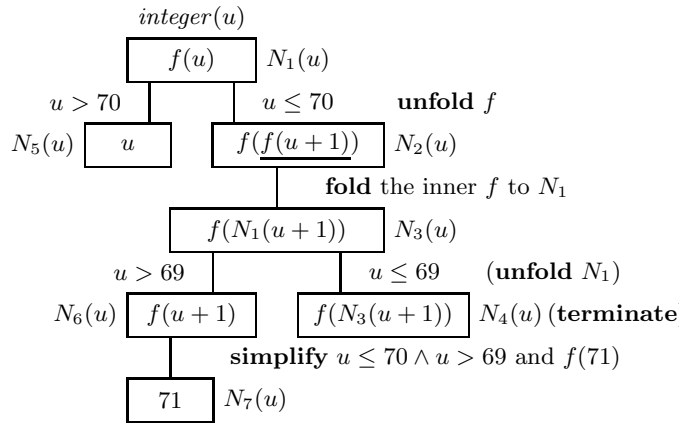

Fig. 4. GPC of 71-function

specific to GPC. Execution of programs based on partial evaluation has two phases [7,11,17], i.e. preprocessing (partial evaluation) and remaining processing (final evaluation). The first phase should evaluate as many portions of a source program as possible in order to save time during the final evaluation. However, it should avoid working on portions of the program not needed in the final evaluation in order to save the partial evaluation time. Moreover, it should terminate because it is a preprocessor. The termination can be guaranteed by setting up maximum execution time and residual program size. Errors concerning partial evaluation are:

1. Omission Error: We sometimes miss to evaluate a part of a source program that can be evaluated at the partial evaluation time in principle. This spoils the efficiency of a residual program.
2. Commission Error: We sometimes evaluate a part of a source program that is not evaluated in total computation. This causes a longer partial evaluation time and a larger residual program. This can also lead to transformation-time errors, such as **if** *too-long-to-prove-but-always-true-predicate* **then** e **else** $1/0$ (division by zero).

Discussions here are not very formal but useful, we believe. Among the two errors above, the commission error is more dangerous. The termination conditions of unfolding a recursive call which we proposed in [7,23] tend to commit too much. They are:

(i) The range of the actual parameter $k(u)$ of a recursive call, e. g. $f(k(u))$, is the same as the one when f was called before as a (direct or indirect) caller of this $f(k(u))$ in an ancestor node of a GPC tree.
(ii) The range of the actual parameter $k(u)$ of a recursive call, e. g. $f(k(u))$, is not included in the domain of f.

If condition (i) holds, then the GPC of the recursive call repeats the same computation as before. If condition (ii) holds, then the GPC of the recursive call does not make sense. It is not very difficult to see why we terminate GPC upon the two conditions.

3.1 New Termination Conditions

Here, we add two new termination conditions. Assume that function f has q parameters $x_1, \ldots, x_q$. Let the immediate caller of a recursive call $f(d_1)$ be $f(d_2)$ where d_1 and d_2 are expressions which do not include any recursive call to f. Let the ranges of d_1 and d_2 be $A = A'_1 \times \cdots \times A'_q$ and $B = B'_1 \times \cdots \times B'_q$, respectively. Note that each component of the product corresponds to the range of a variable x_i. Let the *recursion condition of* $f(d_2)$ *wrt* $f(d_1)$ stands for a condition in f that caused the invocation of $f(d_1)$ from the GPC of $f(d_2)$. Let x be a variable appearing in a simplified recursion condition of $f(d_2)$ wrt $f(d_1)$. Furthermore, let A' be an infinite component of A that is the range of variable x and let B' be corresponding component of B. If there exists A' in A, then the two new termination conditions (iii) and (iv) are:

(iii) A' is not a proper subset of B'.
(iv) Set difference $B' \setminus A'$ is finite.

Since the empty set is finite, conditions (i) and (iv) are not independent. Conditions (ii) and (iii) are not independent either because the domain of a function $f(u)$ and the range of its actual parameter u is equal. Therefore, we check the conditions from (i) to (iv) in that order. These heuristics mean that the new domain of the recursive call is not small enough to become finite in the future. Of course, this is not always true. However, our experiments show that the new termination conditions prevent commission errors very well.

Here, we show five examples in order to look at the effectiveness of the new conditions. Without the new conditions, WSDFU would produce larger, but not more optimized residual programs for the examples. Especially for Example 3.2 to Example 3.4, it would not terminate and produced infinite residual programs.

Example 3.1 Let B be the set of all non-negative integers and A be the set of all positive integers. Let

$$f(u) \equiv \textbf{if } u \leq 1 \textbf{ then } 1 \textbf{ else } f(u-1) + f(u-2).$$

Assume that f is defined on B. Here, the range of $u-1$ and $u-2$ are A and B for $u > 1$, respectively. Then, $f(u-2)$ satisfies condition (i). Since A is infinite but the difference $B \setminus A$ is finite (i.e. $\{0\}$), then $f(u-1)$ satisfies (iv). Therefore, no unfolding occurs here and the residual program produced is the same as the source program. From the residual program, we can solve the recursion equation by an algebraic manipulation system [20] to get a closed form below. This gives us $O(\log(u))$ algorithm for $f(u)$.

$$f(u) = \frac{\phi^u - \hat{\phi}^u}{\sqrt{5}} \quad \text{where } \phi = \frac{1+\sqrt{5}}{2} \text{ and } \hat{\phi} = \frac{1-\sqrt{5}}{2}$$

Example 3.2 Let

$$f(m,n) \equiv \textbf{if } mod(m,n) = 0 \textbf{ then } f(m/n, n+1) + 1 \textbf{ else } 1$$

for non-negative integer m and positive integer n. Let $D_m = \{\text{non-negative integers}\}$. Now, we think about GPC of f wrt $n = 2$. Substituting n by 2, we get a new function

$$f(m,2) \equiv \textbf{if } mod(m,2) = 0 \textbf{ then } f(m/2,3) + 1 \textbf{ else } 1.$$

Since variable n does not appear in recursion condition $mod(m,2)$, we just consider $m/2$ on checking the termination of unfolding $f(m/2,3)$. Since $mod(m,2) = 0$ for $f(m/2,3)$, the range of $m/2$ is D_m. Therefore, $f(m/2,3)$ satisfies condition (iv) and GPC stops unfolding (Here, condition (i) does not apply because $D_m \times \{2\}$ and $D_m \times \{3\}$ are not equal). The residual program produced here is the same as the source program. This means that WSDFU does not spoil the source.

Example 3.3 Let

$$f(m,n) \equiv \textbf{if } m = 0 \wedge n = n \textbf{ then } n \textbf{ else } f(m-1, 3n)$$

for non-negative integers m and n. Then the simplified recursion condition of $f(m,n)$ wrt $f(m-1,3n)$ is $m \neq 0$. Therefore, we just check the ranges of m (i.e. non-negative integer) in $f(m,n)$ and $m-1$ (i.e. non-negative integer) in $f(m-1,3n)$. Then find that the termination condition (iv) holds for $f(m-1,3n)$. The residual program produced here is the same as the source program except that the recursion condition is simplified. The same as for Example 3.1, by using an algebraic manipulation system, we produce the final residual program $f(m,n) \equiv 3^m n$. This gives us $O(\log(m))$ algorithm for $f(m,n)$.

Example 3.4 Let $h(u)$ be the Hailstorm function defined on positive integers:

$$\begin{array}{l} h(u) \equiv \\ \quad \textbf{if } u = 1 \textbf{ then } 1 \\ \quad \textbf{else if } odd(u) \textbf{ then } h(3u+1) \\ \quad \textbf{else } h(u/2). \end{array}$$

Here, $h(u/2)$ satisfies the conditions (i) because $range(u/2) = \{\text{positive integers}\}$ for $u > 1 \wedge even(u)$. While, $h(3u+1)$ does not satisfy any of the termination conditions and the call is unfolded. Note here that $range(3u+1) = \{10, 16, 22, 28, \ldots\}$ and $range((3u+1)/2) = \{5, 8, 11, 14, \ldots\}$. Therefore, the call $h((3u+1)/2)$ satisfies (iii) and GPC terminates. See Fig. 5 for the complete GPC process. The residual program is

$$\begin{array}{l} N_1(u) \equiv \\ \quad \textbf{if } u = 1 \textbf{ then } 1 \\ \quad \textbf{else if } even(u) \textbf{ then } N_1(u/2) \\ \quad \textbf{else } N_1((3u+1)/2). \end{array}$$

The residual program is a little bit more efficient than the source program.

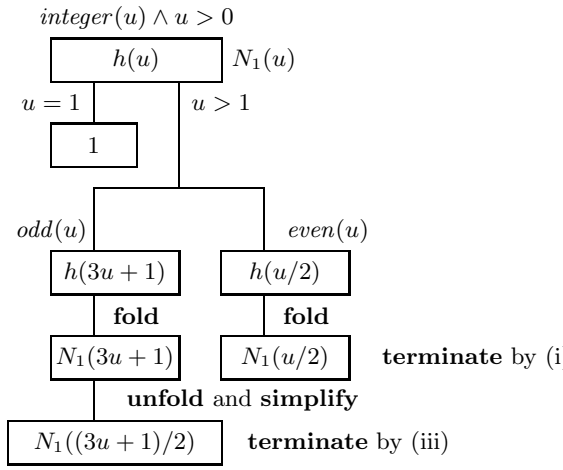

Fig. 5. GPC of Hailstorm function

Example 3.5 Here, we think about GPC of $mod(exp(m, n), d)$ where

$exp(m, n) \equiv$
  if $n = 0$ **then** 1
  else if $odd(n)$ **then** $m * sqr(exp(m, (n - 1)/2))$
  else $sqr(exp(m, n/2))$.

The drawback of the source program is to compute $exp(m, n)$ that cannot be tolerated when m and n are very large, say more than 100 digits each. Our residual program computes the result just using numbers less than d^2. The complete GPC process is shown in Fig. 6. We use the mathematical knowledge $mod(x * y, d) = mod(mod(x, d) * mod(y, d), d)$ in the transformation. The residual program is:

$N_1(m, n, d) \equiv$
  if $n = 0$ **then** 1
  else if $odd(n)$ **then** $mod(mod(m, d)*mod(sqr(N_1(m, (n-1)/2, d)), d), d)$
  else $mod(sqr(N_1(m, n/2, d)), d)$.

We think this program is almost optimal for our purpose. Note that if we do not have condition (iv), $N_1(m, n/2, d)$ in node N_3 is unfolded and the size of the residual program is more than doubled without any remarkable efficiency gain.

4 Theorem Prover

We have tuned up the classical TPU [4] theorem prover because it is powerful and the Lisp implementation is easy to modify. All goto's in the original program

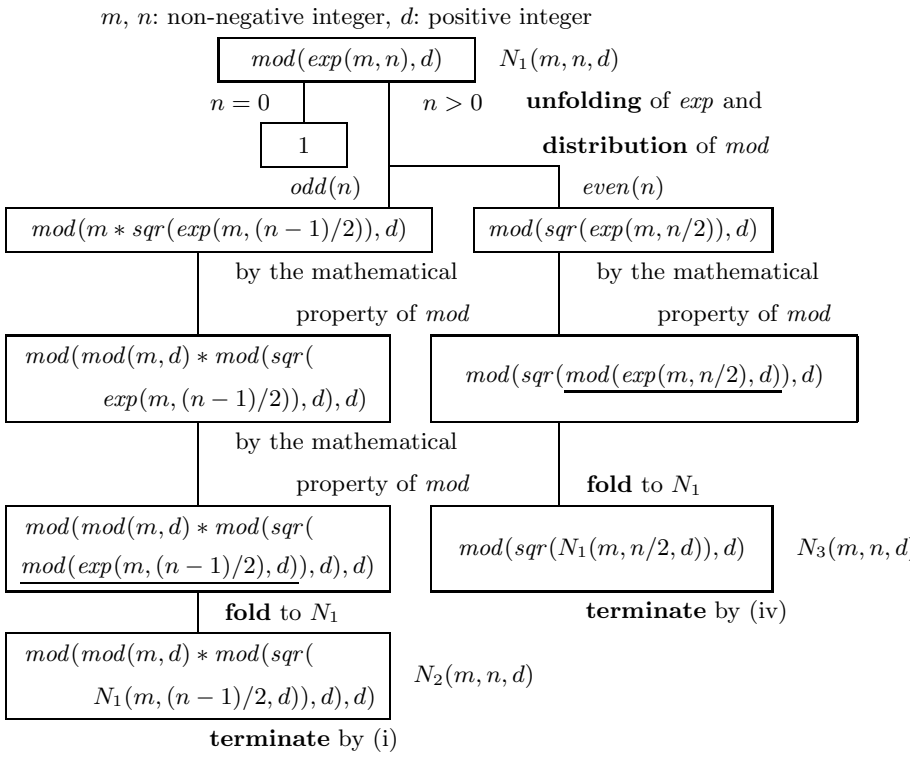

Fig. 6. GPC of $mod(exp(m, n), d)$

have been removed because we want to partially evaluate TPU by GPC in the future. TPU is a uniform unit resolution theorem prover with set-of-support strategy, function depth test and subsumption test. We strengthened TPU with paramodulation so that theorems including equalities could be dealt with easily. Furthermore, term rewriting facility was added to the prover. Before submitting to the prover, theorems are transformed to simpler or shorter form by rewriting rules. For example, atomic formula $x + 1 > 6$ is transformed to $x > 5$.

The theorem prover plays the most important role in WSDFU (Fig. 7). The unfolding or folding of a recursive call terminates on the failure of proving a theorem (i.e., checking the termination or folding conditions) in a predetermined time [18,19]. This may cause omission errors but we'd be rather afraid of commission errors.

5 Benchmark Tests

This section describes 15 source programs and their residual programs produced automatically by WSDFU. All the residual programs are almost optimal, we

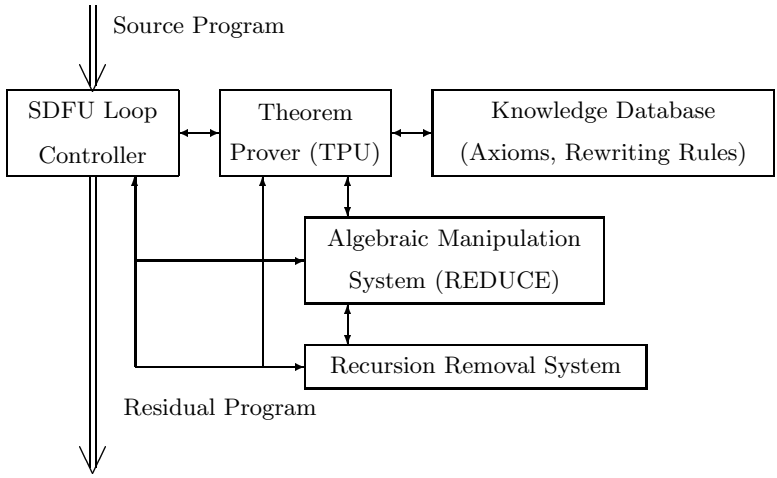

Fig. 7. Structure of WSDFU

believe. Every residual program was produced in less than 100 seconds on a notebook PC with Pentium II processor 366MHz and Allegro Common Lisp 5.0.1 for Windows 98. Although we have dealt with more test programs, those shown here are representatives and were chosen based on their simplicity and inefficiency. If a program is already optimal, it is impossible for WSDFU to produce a better one.

Here, we have three kinds of programs: In order to get almost optimal residual programs (1) WSDFU does not require any knowledge from users (5.1–5.10). (2) WSDFU needs some knowledge concerning source and subprograms from users (5.11–5.12). (3) WSDFU extensively uses an algebraic manipulation system and a recursion removal system (5.13–5.15).

Although, Lisp is the source language of the GPC system, here we use a more compact Pascal-like language to describe programs. Lists are written in Prolog style. For example, $[\,]$, $[a]$, $[a,b]$ and $[a,b|x]$ stand for NIL, $cons(a, NIL)$, $cons(a, cons(b, NIL))$ and $cons(a, cons(b, x))$, respectively.

5.1 Towers of Hanoi Problem (*mvhanoi*)

The source program $move(hanoi(n,a,b,c),n,m)$ is a naive version of computing the m-th move of 2^n-1 moves to solve n disk Towers of Hanoi problem. Auxiliary functions *hanoi* and *move* are given below:

$hanoi(n,a,b,c) \equiv$
  if $n = 1$ **then** $[[a,c]]$
  else $[hanoi(n-1,a,c,b), [a,c] | hanoi(n-1,b,a,c)]$,
$move(s,n,m) \equiv$
  if $n = 1$ **then** $car(s)$

Table 1. Results of benchmark tests. The rightmost column shows the number of times that the theorem prover was called during GPC of a given program. The time shown is the overall transformation time used by GPC.

#	Program	Time in sec.	TPU call
5.1	*mvhanoi*	61.4	83
5.2	*mvhanoi16*	9.5	34
5.3	*mvhanoi16a*	8.8	32
5.4	*mvhanoi3*	43.5	103
5.5	*mvhanoi3a*	28.2	79
5.6	*allonetwo*	1.4	11
5.7	*lengthcap*	5.3	30
5.8	*revapp1*	2.9	22
5.9	*revapp*	6.0	40
5.10	*matchaab*	20.2	126
5.11	*revrev*	1.5	11
5.12	*modexp*	62.8	62
5.13	*lastapp*	4.4	31
5.14	*f71*	6.1	29
5.15	*m91*	94.0	305

else if $m = 2^{n-1}$ **then** $car(cdr(s))$
else if $m > 2^{n-1}$ **then** $move(cdr(cdr(s)), n-1, m-2^{n-1})$
else $move(car(s), n-1, m)$.

The residual program is:

$N_1(n, m, a, b, c) \equiv$
if $n = 1$ **then** $[a, c]$
else if $m = 2^{n-1}$ **then** $[a, c]$
else if $m > 2^{n-1}$ **then** $N_1(n-1, m-2^{n-1}, b, a, c)$
else $N_1(n-1, m, a, c, b)$.

While the source is $O(2^n)$, the residual is $O(n)$ if we assume computation of 2^n costs $O(1)$.

5.2 Towers of Hanoi Problem (*mvhanoi16*)

We specialize $move(hanoi(n, a, b, c), n, m)$ wrt $m = 16$, here. The source program is $move(hanoi(n, a, b, c), n, 16)$. The residual is $N_1(n, a, b, c)$ below. The same as 5.1, while the source is $O(2^n)$, the residual is $O(n)$.

$N_1(n, a, b, c) \equiv$ **if** $5 = n$ **then** $[a, c]$ **else** $N_1(n-1, a, c, b)$.

5.3 Towers of Hanoi Problem (*mvhanoi16a*)

We want to perform the same specialization as 5.2 but we use the residual program of 5.1 instead of using naive *hanoi* and *move* programs. Therefore, the source program is $f(n, 16, a, b, c)$ where

$f(n, m, a, b, c) \equiv$
if $n = 1$ **then** $[a, c]$
else if $m = 2^{n-1}$ **then** $[a, c]$
else if $m > 2^{n-1}$ **then** $f(n-1, m-2^{n-1}, b, a, c)$
else $f(n-1, m, a, c, b)$.

The residual program is identical to that of 5.2 but produced in a shorter time.

5.4 Towers of Hanoi Problem (*mvhanoi3*)

We specialize $move(hanoi(n, a, b, c), n, m)$ wrt $n = 3$, here. The source program is $move(hanoi(3, a, b, c), 3, m)$ and the residual program is $N_1(m, a, b, c)$ below. While the source is $O(2^3)$, the residual is $O(3)$.

$N_1(m, a, b, c) \equiv$
if $m = 4$ **then** $[a, c]$
else if $m > 4$ **then**
if $m = 6$ **then** $[b, c]$
else if $m > 6$ **then** $[a, c]$
else $[b, a]$
else if $m = 2$ **then** $[a, b]$
else if $m > 2$ **then** $[c, b]$
else $[a, c]$.

5.5 Towers of Hanoi Problem (*mvhanoi3a*)

We want to perform the same specialization as 5.4 but we use the residual program of 5.1 instead of using naive *hanoi* and *move* programs. The same as 5.3, we obtain the same residual program as 5.4 but in a shorter time.

5.6 Elimination of Intermediate Data (*allonetwo*)

Source program $allones(alltwos(x))$ replaces every element of a given list by 2 and then replaces every 2 by 1. The residual program $N_1(x)$ directly replaces every element of a given list by 1. The efficiency gain here is that the residual does not produce an intermediate list consisting of 2'.

$allones(x) \equiv$ **if** $Null(x)$ **then** $[\,]$ **else** $[1|allones(cdr(x))]$,
$alltwos(x) \equiv$ **if** $Null(x)$ **then** $[\,]$ **else** $[2|alltwos(cdr(x))]$,
$N_1(x) \equiv$ **if** $Null(x)$ **then** $[\,]$ **else** $[1|N_1(cdr(x))]$.

5.7 Elimination of Intermediate Data (*lengthcap*)

Source program $length(cap(x, y))$ makes the intersection of two given lists x and y and then scan the intersection for its length. The residual program $N_1(x, y)$ does not make the intersection itself but just counts its length.

$length(x) \equiv$ **if** $Null(x)$ **then** 0 **else** $1 + length(cdr(x))$,
$cap(x, y) \equiv$
  if $Null(x)$ **then** $[\,]$
  else if $Member(car(x), y)$ **then** $[car(x)|cap(cdr(x), y)]$
  else $cap(cdr(x), y)$,
$N_1(x, y) \equiv$
  if $Null(x)$ **then** 0
  else if $Member(car(x), y)$ **then** $1 + N_1(cdr(x), y)$
  else $N_1(cdr(x), y)$.

5.8 Elimination of Intermediate Data (*revapp1*)

Source program $rev(app(x, [y]))$ appends a unit list $[y]$ to a given list x then reverses the appended list. The residual program $N_1(x, y)$ produces the same results without building the intermediate appended list. Program rev is a naive reverse using append.

$rev(x) \equiv$ **if** $Null(x)$ **then** $[\,]$ **else** $app(rev(cdr(x)), [car(x)])$,
$app(x, y) \equiv$ **if** $Null(x)$ **then** y **else** $[car(x)|app(cdr(x), y)]$,
$N_1(x, y) \equiv$ **if** $Null(x)$ **then** $[y]$ **else** $app(N_1(cdr(x), y), [car(x)])$.

5.9 Elimination of Intermediate Data (*revapp*)

Source program $rev(app(x, y))$ appends a list y to a list x then reverse the appended list. The residual program $N_1(x, y)$ produces the same results without building the intermediate appended list.

$N_1(x, y) \equiv$ **if** $Null(x)$ **then** $N_2(x, y)$ **else** $app(N_1(cdr(x), y), [car(x)])$,
$N_2(x, y) \equiv$ **if** $Null(y)$ **then** $[\,]$ **else** $app(N_2(x, cdr(y)), [car(y)])$.

5.10 Pattern Matcher (*matchaab*)

Source program $matchaab(x)$ is a non-linear pattern matcher that check if there is pattern $[a, a, b]$ in a given text x. The residual program is a KMP-type linear pattern matcher. Note that the pattern matcher used here in the source program is as naive as the one in [6,14,22], but more naive than the one used in [1,7]. The generation of a Boyer-Moore type pattern matcher is discussed in [1,13].

$matchaab(x) \equiv f([a, a, b], x, [a, a, b], x)$,
$f(p, t, p_0, t_0) \equiv$
  if $Null(p)$ **then** *true*
  else if $Null(t)$ **then** *false*
  else if $car(p) = car(t)$ **then** $f(cdr(p), cdr(t), p_0, t_0)$
  else if $Null(t_0)$ **then** *false*
  else $f(p_0, cdr(t_0), p_0, cdr(t_0))$,

$N_1(x) \equiv$
  if $Null(x)$ **then** *false*
  else if $a = car(x)$ **then**
    if $Null(cdr(x))$ **then** *false*
    else if $a = cadr(x)$ **then** $N_8(x)$
    else $N_9(x)$
  else $N_5(x)$,
$N_5(x) \equiv N_1(cdr(x))$,
$N_8(x) \equiv$
  if $Null(cddr(x))$ **then** *false*
  else if $b = caddr(x)$ **then** *true*
  else if $a = caddr(x)$ **then** $N_8(cdr(x))$
  else $N_9(cdr(x))$,
$N_9(x) \equiv N_5(cdr(x))$.

5.11 List Reversal (*revrev*)

Source program $rev(rev(x))$ reverses a given list twice. The residual program $N_1(x)$ just copy a given list x. This time, we gave knowledge about *rev* and *append*, i.e. $rev(append(u, v)) = append(rev(v), rev(u))$ to WSDFU before starting GPC. The residual program N_1 below is equivalent to *copy* function.

$N_1(x) \equiv$ **if** $Null(x)$ **then** $[\,]$ **else** $[car(x)|N_1(cdr(x))]$.

5.12 Example 3.5 in Section 3 (*modexp*)

See Example 3.5 in Section 3 for the problem, source and residual programs.

5.13 Last Element of an Appended List (*lastapp*)

Source program $last(app(x, [y]))$ appends a unit list $[y]$ to a given list x and scan the appended list for the last element. The final residual program $N_1(x, y)$ directly returns y. This time we used a recursion removal system after getting a usual residual program to produce the final residual program.

$last(x) \equiv$ **if** $Null(cdr(x))$ **then** $car(x)$ **else** $last(cdr(x))$,
$N_1(x, y) \equiv$ **if** $Null(x)$ **then** y **else** $N_1(cdr(x), y)$
  $\equiv y$ (by recursion removal [19]).

5.14 71-function (*f71*)

See 71-function example and Fig. 4 in Section 2 for the problem, source and residual programs.

5.15 McCarthy's 91-function (*m91*)

Source program $f(x)$ is McCarthy's 91-function that is a complex double recursive function. The final residual program $N_1(x)$ is $O(1)$. We used a recursion removal system after getting a usual residual program to produce the final one.

$f(x) \equiv \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } f(f(x + 11))$.

First, GPC produces the following long residual program.

$N_1(x) \equiv \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } N_3(x)$,
$N_3(x) \equiv$
  if $x > 89$ **then**
    if $x > 99$ **then** 91
    else if $x > 98$ **then** 91
    else if $x > 97$ **then** 91
    else if $x > 96$ **then** 91
    else if $x > 95$ **then** 91
    else if $x > 94$ **then** 91
    else if $x > 93$ **then** 91
    else if $x > 92$ **then** 91
    else if $x > 91$ **then** 91
    else if $x > 90$ **then** 91
    else 91
  else $f(N_3(x + 11))$
  $\equiv \textbf{if } x > 89 \textbf{ then } 91 \textbf{ else } f(N_3(x + 11))$ (by simplification)
  $\equiv f^{(1+\lfloor(89-x)/11\rfloor)}(91)$ (by recursion removal [19])
  $\equiv 91$. (by simplification)

Finally, the following residual program has been obtained.

$N_1(x) \equiv \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } 91$.

6 Future Works

In the previous section, we have shown 15 GPC examples. Their residual programs are almost optimal. As shown in Table 1, transformation time is quite fast. However, we have not implemented such powerful program transformation methods as generalization of domains of functions, λ-abstraction and tupling. The methods are consistent with WSDFU and it is not a difficult task to build them in our current system. The rest of this section describes the idea of extending WSDFU to include generalization, λ-abstraction and tupling.

6.1 Generalization of Function Domain

As shown above, folding is one of the most useful operations in program transformation [2,3]. Without this operation, we will not produce an optimal residual

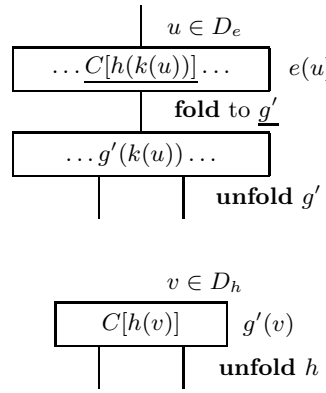

Fig. 8. Generalization and GPC Forest

program from a source program very often. A function is easier to be folded to if it has a larger domain. Therefore, when we define an auxiliary function (i.e. a new node in GPC tree) for residual programs, it is better for folding to extend the domain of the function as much as possible. Generalization in program transformation was first dicussed by Burstall and Darlington [3] and in partial evaluation by Turchin [24]. The generalization is discussed in detail in [21]. However, the extension of the domain is opposite to the specialization of programs. Therefore, we have to do both specialization and generalization in at the same time (or non-deterministically) to produce optimal residual programs. In our current implementation, the domain of a new function defined by WSDFU is given by the environment corresponding to the function. This domain is the most specialized one for the function in some sense. The outline of the modification to the current WSDFU to include a generalization strategy is described below.

Let e be a node in a GPC tree, $e(u)$ be an expression in node e and D_e be the range of variable u at e (see Fig. 8).

First, we choose one subexpression $g(u)$ of $e(u)$ non-deterministically that includes $h(k(u))$ for a primitive function k and a recursive function h. Let $g(u) = C[h(k(u))]$, $g'(v) = C[h(v)]$ and the domain of h be D_h. Since $k(D_e) \subseteq D_h$, g' is a generalized version of g in some sense. Therefore, our generalization strategy is to perform GPC on $g'(v)$ instead of $g(u)$ (see Fig. 8). We add a GPC tree of $g'(v)$ to an existing GPC forest. If $g'(v)$ becomes a recursive function through a successful folding during its GPC process, generalization process is successful and $C[h(k(u))]$ in node e is replaced by $g'(k(u))$. In this case, we call g' the generalization of g. Otherwise, the generalization is unsuccessful and a GPC tree of $g'(v)$ is discarded.

6.2 Tupling and λ-abstraction

Tupling and λ-abstraction factorize similar computations in a source program and avoid to execute them more than once. The techniques were first used explicitly in program transformation in [3]. We can perform them in the simplification phase of WSDFU as follows.

Let $h_1(k_1(u))$ and $h_2(k_2(u))$ be procedure calls to recursive functions h_1 and h_2 in a program $e(u)$, and k_1 and k_2 be primitive functions. Moreover, assume that the domains of h_1 and h_2 are identical. Then we check the following two cases:

1. **λ-abstraction**: If $h_1 = h_2$ and $k_1 = k_2$, then replace all $h_i(k_i(u))$ in $e(u)$ by a fresh variable y that does not appear in $e(u)$ and let the new program be $e'(u)$. Then replace $e(u)$ by $(\lambda y.e'(u))h_1(k_1(u))$. This corresponds to the insertion of **let**-expressions in Lisp.
2. **Tupling**: If $h_1 \neq h_2$ or $k_1 \neq k_2$, then let $h(u) \equiv cons(h_1(k_1(u)), h_2(k_2(u)))$ and check the following two cases:
   2.1 If $h(u)$ can be folded to any ancestor node, say $h'(u)$, then fold $h(u)$ to it (See node N_5 in Fig. 10).
   2.2 If we can find a generalization of $h(u)$, say $h'(v)$, in a predetermined time, then perform the following operation: replace all $h_1(k_1(u))$ and $h_2(k_2(u))$ by $car(y)$ and $cdr(y)$ for a fresh variable y and let the new program be $e'(u)$. Note here that $h(u) = h'(k(u))$ for a primitive function k. Then replace $e(u)$ by $(\lambda y.e'(u))(h'(k(u)))$. If we cannot find a generalization of $h(u)$, then stop tupling (i.e. tupling is unsuccessful).

If the residual program of $h(u)$ itself is recursive, then $h'(u) \equiv h(u)$ in 2.2. Even when there are more than two functions involved, we can perform tupling the same way as above.

6.3 Example 6.1: List of Ascending Factorial Numbers

The source program $afl(n)$ lists factorial numbers from 0! to n!. The drawback of the program is to compute factorials $n+1$ times and costs $O(n^2)$ multiplications. The residual program $new\_afl(n)$ computes $n!$ just once and costs $O(n)$ multiplications (see Fig. 9 and Fig. 10 for a complete GPC).

$afl(n) \equiv$ **if** $n = 0$ **then** $[1]$ **else** $app(afl(n-1), [fact(n)])$,
$fact(n) \equiv$ **if** $n = 0$ **then** 1 **else** $n * fact(n-1)$,
$new\_afl(n) \equiv cdr(h'(n))$,
$h'(n) \equiv [fact(n)|afl(n)] \equiv$
  if $n = 0$ **then** $[1|[1]]$
  else $(\lambda y.((\lambda z.[z|app(cdr(y), [z])])(n * car(y))))(h'(n-1))$

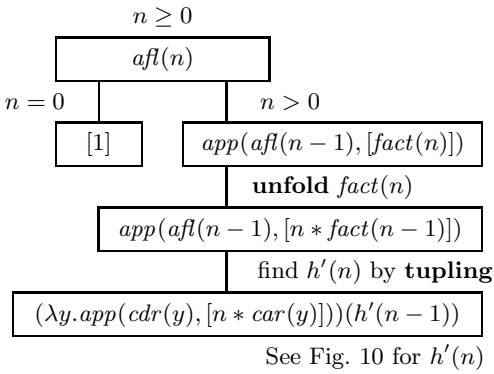

Fig. 9. Finding $h(n) \equiv [fact(n-1)|afl(n-1)]$ by tupling

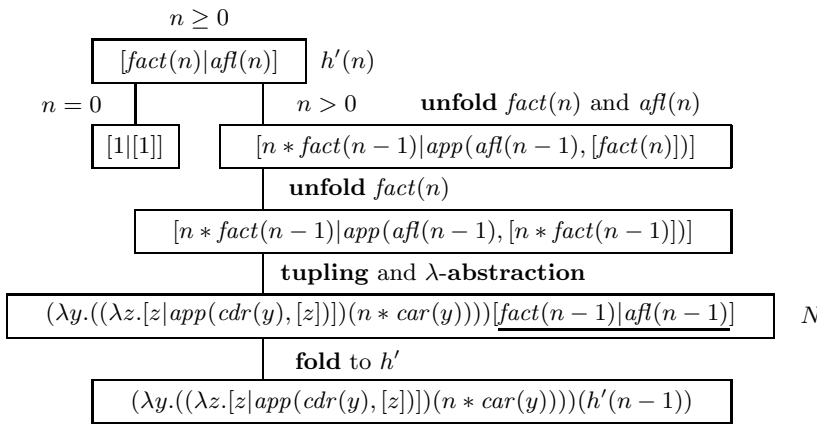

Fig. 10. GPC of $h'(n) \equiv [fact(n)|afl(n)]$

7 Conclusion

We have described GPC and WSDFU, a program transformation method using theorem proving combined with classical partial evaluation, and shown the results of a several benchmark tests using our present implementation. From the benchmark tests shown here, we can conclude that GPC is not far from being applicable to more practical problems. Also, we know how to strengthen WSDFU as described in Section 6.

We can say that our transformation method captures mathematical knowledge and formal reasoning. It allows to express source programs in a declarative, often inefficient style, and then to derive optimized versions exploiting that knowledge. As such, GPC can play an important role in different phases of software development.

A challenging problem is to implement a self-applicable GPC so that we can automatically generate generating extensions and compiler-compilers [10,11].

Acknowledgments

This research was partially supported by the Research for the Future Program of JSPS and New Technology Development Project. The authors are grateful to Torben Mogensen for valuable comments and discussions about the manuscript of this paper.

References

1. Amtoft, T., Consel, C., Danvy, O., Malmkjaer, K.: The abstraction and instantiation of string-matching programs. BRICS RS-01-12, Department of Computer Science, University of Aarhus (2001)
2. Bird, R. S.: Improving programs by the introduction of recursion. Comm. ACM **20** (11) (1977) 856–863
3. Burstall, R. M., Darlington, J. A.: Transformation System for Developing Recursive Programs. JACM **24** (1) (1977) 44–67
4. Chang, C., Lee, R. C.: Symbolic Logic and Mechanical Theorem Proving. Academic Press (1973)
5. Danvy, O., Glück, R., Thiemann, P. (eds.): Partial Evaluation. Proceedings. Lecture Notes in Comp. Science **1110** Springer-Verlag (1996)
6. Futamura, Y., Nogi, K.: Generalized partial computation. in Bjørner, D., Ershov, A. P., Jones, N. D. (eds.): Partial Evaluation and Mixed Computation. North-Holland, Amsterdam (1988) 133–151
7. Futamura, Y., Nogi, K., Takano, A.: Essence of generalized partial computation. Theoretical Computer Science **90** (1991) 61–79
8. Futamura, Y., Nogi, K.: Program Transformation Based on Generalized Partial Computation. 5241678, US-Patent, Aug. 31, 1993
9. Futamura, Y., Otani, H.: Recursion removal rules for linear recursive programs and their effectiveness. Computer Software, JSSST **15** (3) (1998) 38–49 (in Japanese)
10. Futamura, Y.: Partial Evaluation of Computation Process — An approach to a Compiler-Compiler. Higher-Order and Symbolic Computation **12** (4) (1999) 381–391
11. Futamura, Y.: Partial Evaluation of Computation Process Revisited. Higher-Order and Symbolic Computation **12** (4) (1999) 377–380
12. Futamura, Y., Konishi, Z., Glück, R.: Program Transformation System Based on Generalized Partial Computation. New Generation Computing **20** (1) (2001) 75–99
13. Futamura, Y., Konishi, Z., Glück, R.: Automatic Generation of Efficient String Matching Algorithms by Generalized Partial Computation. ASIA-PEPM 2002 (2002)

14. Glück, R., Klimov, A. V.: Occam's razor in metacomputation: the notion of a perfect process tree. In: Cousot, P., et al. (eds.): Static Analysis. Lecture Notes in Comp. Science **724** Springer-Verlag (1993) 112–123
15. Graham, R. L., Knuth, D. E., Patashnik, O.: Concrete Mathematics. Addison-Wesley (1989)
16. Hearn, A. C.: REDUCE — A Case Study in Algebra System Development. Lecture Notes in Comp. Science **144** Springer-Verlag, Berlin (1982)
17. Jones, N. D.: An Introduction to Partial Evaluation. ACM Computing Surveys **28** (3) (1996) 480–503
18. Konishi, Z., Futamura, Y.: A theorem proving system and a terminating process for Generalized Partial Computation (GPC). RIMS Workshop on Program Transformation, Symbolic Computation and Algebraic Manipulation, (1999) 59–64 (in Japanese)
19. Konishi, Z., Futamura, Y.: Recursion removal on generalized partial computation (GPC). Proc. of the 17th National Conference, C5-2, JSSST (2000) (in Japanese)
20. Matsuya, M., Futamura, Y.: Program transformation and algebraic manipulation. RIMS Workshop on Program Transformation, Symbolic Computation and Algebraic Manipulation, (1999) 115–122 (in Japanese)
21. Pettorossi, A., Proietti, M.: Rules and Strategies for Transforming Functional and Logic Programs. ACM Computing Surveys **28** (2) (1996) 360–414
22. Sørensen, M. H., Glück, R., Jones, N. D.: A positive supercompiler. Journal of Functional Programming **6** (6) (1996) 811–838
23. Takano, A., Nogi, K., Futamura, Y.: Termination Conditions of Generalized Partial Computation. Proceedings of the 7th National Conference, C8-1, JSSST (1990) (in Japanese)
24. Turchin, V. F.: The concept of a supercompiler. ACM TOPLAS **8** (3) (1986) 292–325