Partial Evaluation and Mixed Computation
D. Bjørner, A.P. Ershov and N.D. Jones (Editors)
Elsevier Science Publishers B.V. (North-Holland)

GENERALIZED PARTIAL COMPUTATION

Yoshihiko Futamura and Kenroku Nogi

Advanced Research Laboratory
Hitachi, Ltd.
Kokubunji, Tokyo, Japan

In this paper, a new partial computation method that makes use of a theorem prover when evaluating conditions of conditional expressions is proposed. This method is called "Generalized Partial Computation (GPC)" because it is more powerful than conventional methods that use interpreters when evaluating conditions of conditional expressions (The old method is called "Interpreter Dependent Partial Computation (IDPC)" in this paper).
To show GPC's power, it is applied to a program transformation of an O(m*n) time program to an O(m+n) program. Also shown is that the combination of GPC and recursion removal methods is a very powerful program transformation technique. Finally, two more order changing program transformation examples are given.

1. INTRODUCTION

The practical importance of partial computation in computer science was first recognized around 1970. Partial computation has been considered in the following way with this kind of program transformation [2]:

> Let f be a program (function) with two parameters k (a known parameter) and u (an unknown parameter). First, finish all the f computations that can be performed by using only the k value while f computations that cannot be performed without knowing the u value should be left intact. Then we have a new program, f_{k0}, having the property
> $$f_{k0}[u]=f[k0;u]$$
> where $k0$ stands for the k value.

This formula is similar to Kleene's s-m-n theorem [7] as first pointed out by Ershov [5]. However, f_{k0} of the s-m-n theorem is just a function closure λ [[u];f[k0;u]] and it does not improve program efficiency. On the contrary, since computations concerning k have 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 $u0$.

Let α [f;k0] be the result of partially computing f when k=k0, i.e. if α is a partial evaluator, then α [f;k0]=f_{k0}. Let I be a programming language interpreter and p be a program. The following relationships are then well known [2, 3, 12]:

α [I;p] is an object program.
α [α ;I] is a compiler.
α [α ;α] is a compiler-compiler.

By using an interpreter for evaluating conditions of conditional expression, α 's have been implemented. This implementation has a limitation that it uses only the k value. The basic idea of α and its limitation will be discussed in Appendix 1.

In this paper, a new partial evaluator using not only the k-value, but also

all information on the operating environment of a program is proposed. Let β be a new partial evaluator, e a program and i information on the operating environment. The result of partially computing e on i is described as β [e;i] in full, or $(e)_i$ as an abbreviation.
β uses a theorem prover and information i when it evaluates conditions of a conditional expression. Note that, for notational convenience, program e is a form (function with its parameters) for β while a program is a function for α (see Appendix 1).

Let b be a program, and k and u be free variables in b. If $f=\lambda$ [[k;u];b] then α [f;k0]$=\lambda$ [[u];β [b;{k=k0}]] where {k=k0} is a predicate (information) denoting that $k0$ is the k-value. This shows that β can be considered to be more general than α. Therefore, β is called a Generalized Partial Computation (GPC) while α is called an Interpreter Dependent Partial Computation (IDPC) (see Appendix 1).

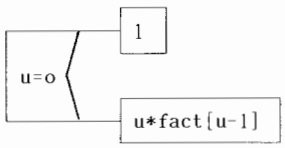
The principle of β is simple and has been used in informal program transformations. It was also implicitly used in Kahn's partial evaluator [6]. This paper extends the principle to a more powerful and systematic partial computation method. First, the basic idea of β and its application to program transformation from a nonlinear pattern matcher to a KMP type linear pattern matcher [8] are shown. This is an example transformation of an O(m*n) time program to an O(m+n) time program. Finally, least fixpoints of recursive programs are produced using a combination of partial computation and recursion removal techniques. These productions are also examples of order changing program transformations.

2. GENERALIZED PARTIAL COMPUTATION PRINCIPLE

Generalized Partial Computation (GPC) has been established by formalizing human informal program transformation processes (getting a fixpoint of a recursive function [9] is an example of the transformation). GPC uses a logic system to evaluate a predicate which is not evaluable for an interpreter . The logic system is consistent with the interpreter and is called underlying logic. Before explaining the basic idea of GPC, definitions will be provided for u-form, u-information and underlying logic. Reader's knowledge about PAD [4] (a graphical representation for structured programs) and LISP M-expression[10] are assumed in the following discourse.

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

Example 1: A conditional u-form written in PAD:



(This means
if u=o *then* 1 *else* u*fact[u-1])

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

value of u-form: Let e be a u-form and *eval* be a fixpoint 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 1: Let a and b be constants or $\perp$. Then $a \sqsubseteq b$ if and only if a=b or a=$\perp$.

u-information: A conjunction of predicates on u (Note that this is a u-form too).

Example 2: Examples of u-information:
$\neg$null[u]$\wedge$(A=car[u])$\wedge$null[cdr[u]]
true (This is represented by ϕ.)

Definition 2: Compatibility with *eval*:
Let i and p be any u-information such that $i \vdash^{*} p$, where $i \vdash^{*} p$ means that p is provable from i based on some logic system. The logic system is compatible with *eval* if and only if eval[p;((u.c))]$\subseteq$true for any constant c such that eval[i;((u.c))]$\subseteq$true.

When eval[p;((u.c))] is always true or false for any u-form (predicate) p and any constant c, then $\subseteq$ is equivalent to $=$. The compatibility property guarantees the soundness of a logic system with respect to the interpreter *eval*.

Example 3: 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*.

underlying logic: The logic system is called underlying logic if and only if it is compatible with eval.

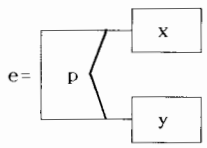
Depending on the predicate evaluation power of eval, an underlying logic can be any logic system, for example propositional logic, predicate logic, or informal logic.

generalized partial computation method β and β-L partial evaluator:
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 β[L;e;i]. 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 β[e;i] in the following discourse means a β-L partial evaluator, and β[e;i]=β[L;e;i] for some L. When β is also clear in the context, $(e)_i$ is used to represent β[e;i].

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

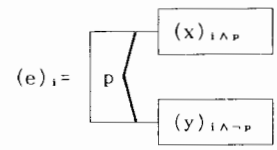
Example 4: Let $\beta 0$ be a transformation such that $\beta 0$[e;i]=e. Then $\beta 0$ is a very trivial partial computation method.

Example 5: Partial computation method $\beta 1$:
(1) If e is a conditional expression, i.e.



(where x and y are u-expressions)

then
(1.1) If $i \vdash^{*} p$, then $(e)_i=(x)_i$
(1.2) If $i \vdash^{*} \neg p$, then $(e)_i=(y)_i$
(1.3) If it is not easy to decide if $i \vdash^{*} p$ or $i \vdash^{*} \neg p$, then



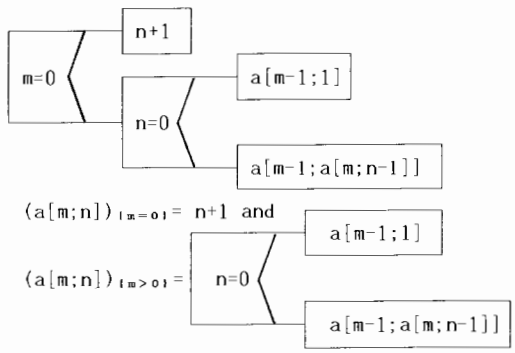
(2) If e is not a conditional expression, then
$(e)_i=e$ (i.e. there is no transformation)

Theorem proving and generation of a new predicate have also been conducted in symbolic execution and program verification [11] as in $\beta 1$. However, they have never had the function of generating a conditional form described in the (1.3) above.

Example 6: Let $\beta 1$ be the $\beta 1$-L0 partial evaluator for *L0* of Example 3. Then $\beta 1[e;\phi]=e$.

Example 7: Partial computation of Ackermann's function when $\beta 1$ uses informal logic on natural numbers:

a[m;n]=

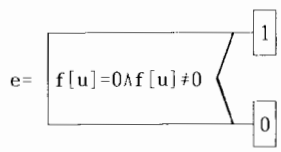


Definition 3: Let d and e be u-forms, and i be u-information. Then $d\subseteq^i e$ if and only if $eval[d;((u.c))]\subseteq eval[e;((u.c))]$ for any constant c such that $eval[i;((u.c))]\subseteq true$.

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

Definition 4: Correctness of β partial evaluator:
β partial evaluator is correct if and only if $e\subseteq^i \beta[e;i]$ for any u-form e and any u-information i.

The correctness of $\beta 0$ partial evaluator and $\beta 1$-L0 partial evaluator is trivial. Let f be an undefined function, and e be a u-form described below:



Then Definition 4 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 5: Correctness of partial computation method β:
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. Two correctness theorems will be given in Appendix 2. Proofs can be conducted by induction on the depth of conditional forms contained in a u-form.

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

Definition 6: Let d and e be u-forms and i be a u-information. $d \equiv^{i} e$ if and only if $d \subseteq^{i} e$ and $e \subseteq^{i} d$.

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

Definition 7: Strict correctness of β partial evaluator:
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 $e \equiv \beta[e;\phi]$. $e \equiv \beta[e;\phi]$ 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. $eval[p;((c.u))]$ is defined for any constant c, then $\beta 1$ partial evaluator is strictly correct.

Note that u-information is a dynamic part of information on the operating environment of a program. It varies during partial computation depending on program structure. On the contrary, information on functions (for example $car[con\ s[x;y]] = x$) 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.

3. MORE PRACTICAL GPC

As described in the previous section, $\beta 0$ 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 e types such as constant, variable, or composite expression. 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 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 repetition of similar computation. A technique called "partial definition" is introduced below to eliminate therepetition as often as possible.

Before starting partial computation of u-form f[g] with u-information i, let f_g^i be a new function name as a result of the partial computation. *f*, *g* and *i* is 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 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 [2]. This is a special case of a general program transformation technique called folding [1]. Introducing a partially defined function is nearly equal to adding the rule $f_g^i[u] <= \beta 2[f[g];i]$ to a system of recursion equations and then continuing program transformation with unfolding $\beta 2[f[g];i]$. The use of a partially defined function is similar to folding $\beta 2[f[g];i]$ to $f_g^i[u]$.

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

Definition 8: Partial definition:
Let *i* be u-information and e=f[g] be 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 9: Partially defined function:
Function f_d^j in Definition 8 is called a partially defined function for $(e)_i$.

Example 7: 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=cdr[u], 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 $i \vdash^* j3/cdr[u]$.

Two partially defined functions f_d^{j2} and f_d^{j3} in the examples above are for $(f[g])_i$ and $j3 \vdash^* j2$. In this case *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*.

Assume that f_d^j is a partially defined function for $(f[g])_i$ and that the underlying logic is substitutive (i.e. if $i \vdash^* j$ then $i/k \vdash^* j/k$), then $(f[g])_i$ can be replaced by $(f_d^j[k])_i$. The very rough explanation for the correctness of this replacement is:
$(f_d^j[k])_i \equiv^i ((f[d])_j/k)_i \equiv^i ((f[d]/k)_{j/k})_i \equiv^i ((f[g])_{j/k})_i \equiv^i (f[g])_i$.

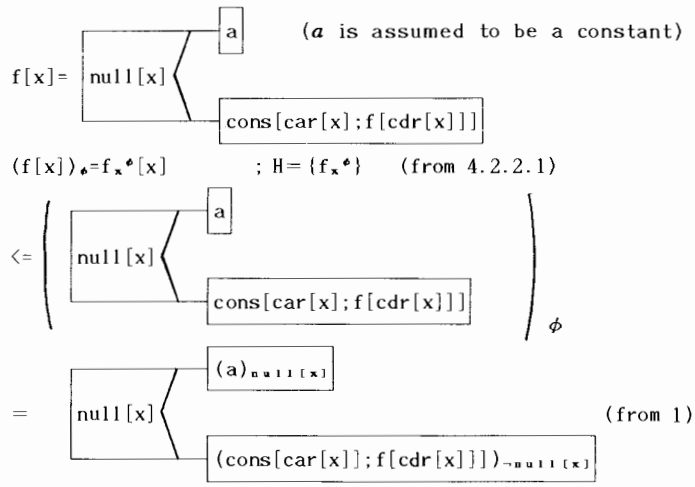
This use of partially defined function f_g^i causes introduction of recursive calls to f_g^i. Therefore, the result of partial computation is a set of recursive functions (for example, pattern matcher h1 and its auxiliary functions h2, h3 and h4 in the next section). This recursion introduction has the following two effects:
(1) It may dramatically increase the effectiveness of partial computation by partially computing the partial result of f_g^i (see the example in a later section).
(2) It may terminate an infinite partial computation caused by repetition of a similar computation.

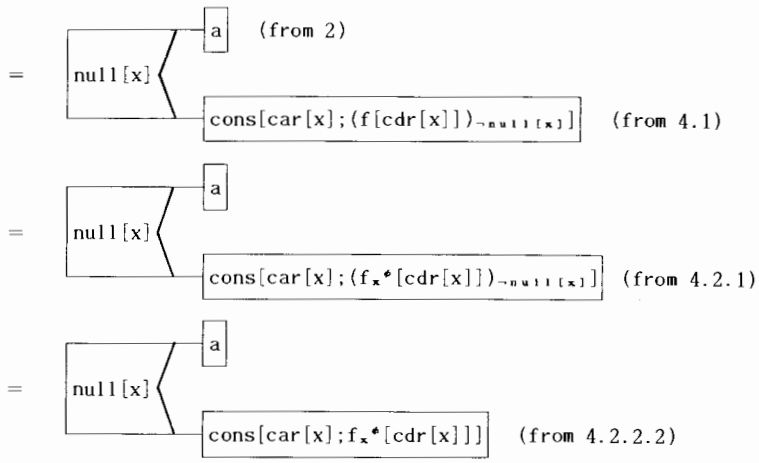
Note that (1) and (2) above are exclusive. When effect (1) is not expected, i.e. when a program will not be improved, the result of partial computation will be too large or partial computation will not terminate. Thus, (2) is expected. Selecting either (1) or (2) is not decidable. However, a practical 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 partial definition operator <=. Let f be a function name. Then f[u]<=$\beta 2$[e;i] (or f[u]<= $(e)_i$) means that when f is referred after the execution of <= 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.

Example 8: 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,
 (4.1) If f is a primitive function such as LISP SUBR, then $(e)_i$=f[$(g)_i$].
 (4.2) If f is a nonprimitive function such as LISP EXPR, then
 let f=λ[[u];b] and:
  (4.2.1) *If $(e)_i$ is partially defined*, then let f_d^j=λ[[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 $(e)_i=(f_d^j[k])_i$.
  (4.2.2) *If $(e)_i$ is not partially defined*, then select one of the following operations (4.2.2.1) or (4.2.2.2) depending on its effectiveness (note that this selection is up to the user of the partial evaluator).
   (4.2.2.1) If it is effective in performing further partial computation, then $(e)_i=f_g^i[u]$; H=H∪{f_g^i};f_g^i[u]<=$(b/g)_i$.
   (4.2.2.2) Otherwise, $(e)_i$=e.

Example 9: Partial computation of append:



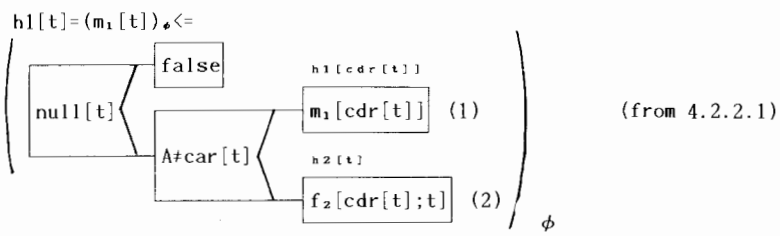


$\beta 2$ terminates when e is a constant or a variable, or when the user of $\beta 2$ decides to terminate. Finding practical methods for automatic termination is an interesting research problem. The correctness of $\beta 2$ will not be discussed in this paper. In the following discussion, f_x^i may be represented by such a simpler symbol as hn. Furthermore, partial definitions may be omitted when partially defined functions will not be used later.

4. PARTIAL COMPUTATION OF A SIMPLE PATTERN MATCHER

As an example showing the behavior of $\beta 2$, a simple nonlinear pattern matcher m[p;t] is partially evaluated with pattern p=(A A A B) using $\beta 2$ described in the previous section. The definition of m[p;t] is given in Appendix 1. The partial computation result is a KMP [8] type linear pattern matcher. This is an example of order changing program transformation.

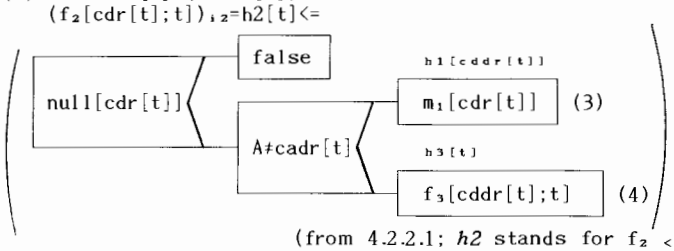
For computation convenience, $m_{(A\ A\ A\ B)}[t]$ is first derived by using α instead of directly applying $\beta 2$ to m (see Appendix 1). Partial definitions which are not used later will not be stated explicitly in the following discourse (for example in (1), (3), (5) and (7) below). Note also that m_1 is an abbreviation for $m_{(A\ A\ A\ B)}$ and informal logic on LISP is used as underlying logic.



where *h1* stands for $m_{1\ t}^{*}$ and forms written in small letters stand for the final result of partial computation with the original forms just beneath them. Numbers such as (1) and (2) written right side of PAD rectangles mean that the partial computation of t-forms contained in the rectangles will be referred later by the numbers.

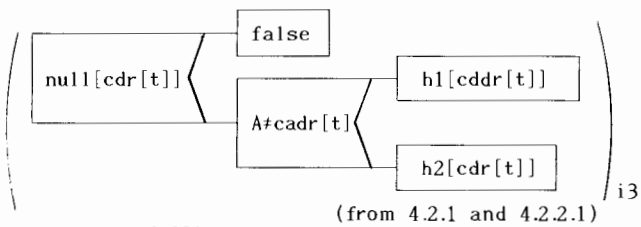
(1) $i1 = \neg null[t] \wedge (A \neq car[t])$
$(m_1[cdr[t]])_{i1} = h1[cdr[t]]$ (from 4.2.1 and 4.2.2.2)

(2) $i2 = \neg null[t] \wedge (A = car[t])$
$(f_2[cdr[t];t])_{i2} = h2[t] \Leftarrow$



(from 4.2.2.1; *h2* stands for $f_{2\langle cdr[t],\,t\rangle^{i2}}$)

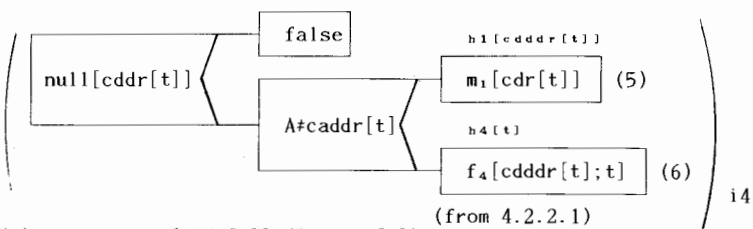
(3) $i3 = i2 \wedge \neg null[cdr[t]] \wedge (A \neq cadr[t])$
$(m_1[cdr[t]])_{i3} =$
$(h1[cdr[t]])_{i3} =$



(from 4.2.1 and 4.2.2.1)

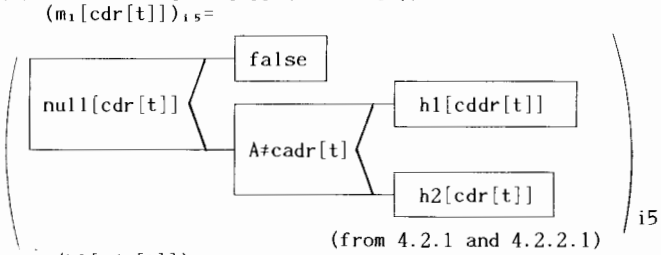
$= (h1[cddr[t]])_{i3}$
$= h1[cddr[t]]$ (from 4.2.1 and 4.2.2.2)

(4) $i4 = i2 \wedge \neg null[cdr[t]] \wedge (A = cadr[t])$
$(f_3[cddr[t];t])_{i4} = h3[t] \Leftarrow$



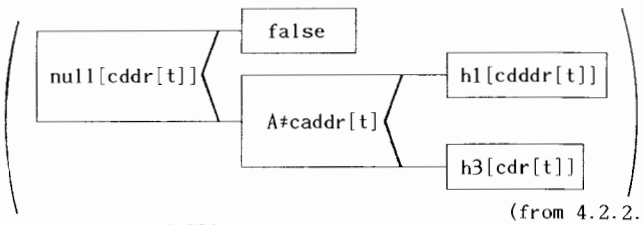
(from 4.2.2.1)

(5) $i5 = i4 \wedge \neg null[cddr[t]] \wedge (A \neq caddr[t])$
$(m_1[cdr[t]])_{i5} =$



(from 4.2.1 and 4.2.2.1)

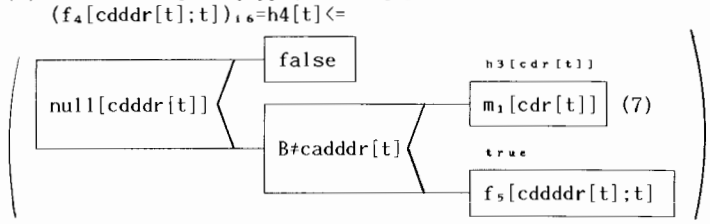
$= (h2[cdr[t]])_{i5} =$



i_5 (from 4.2.2.1)

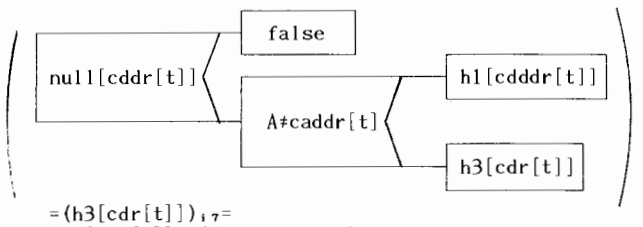
$=(h1[cdddr[t]])_{i_5}$
$=h1[cdddr[t]]$ (from 4.2.2.2)

(6) $i6=i4 \wedge \neg null[cddr[t]] \wedge (A=caddr[t])$
$(f_4[cdddr[t];t])_{i_6}=h4[t]<=$



i_6 (from 4.2.2.1)

(7) $i7=i6 \wedge \neg null[cdddr[t]] \wedge (B \neq cadddr[t])$
$(m_1[cdr[t]])_{i_7}=$
$(h2[cdr[t]])_{i_7}=$ (the same as (5))



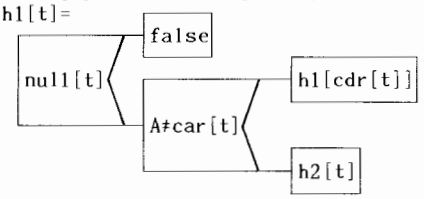
(from 4.2.2.1)

i_7

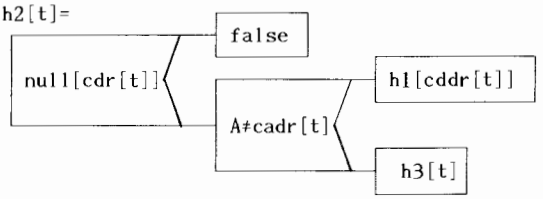
$=(h3[cdr[t]])_{i_7}=$
$=h3[cdr[t]]$ (from 4.2.2.2)

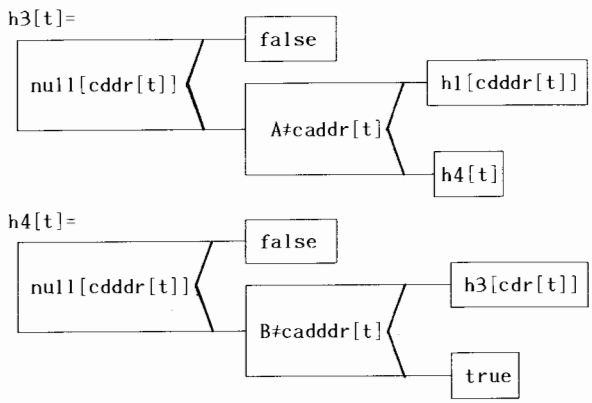
Thus h1[t] has been completely defined as $(m_1[t])_\phi$.

h1[t]=



h2[t]=





Example computation :
h1[(A A B C A A A B)]=h2[(A A B C A A A B)]=h3[(A A B C A A A B)]=
h1[(C A A A B)]=h1[(A A A B)]=h2[(A A A B)]=h3[(A A A B)]=h4[(A A A B)]=true

It is clear from the above example that h1[t] runs in almost the same manner as the KMP pattern matcher. Therefore, if $cd^n r$ can be executed in constant time, h1[t] can run in linear time. Note that the transformation from $(m_1[cdr[t]])_{i7}$ to $(h2[cdr[t]])_{i7}$ in (7) is almost the same as that of $(m_1[cdr[\ t]])_{i5}$ to $(h2[cdr[t]])_{i5}$ in (5). This means that $\beta 2$ fails to avoid repeating the same computation. However, it is not very difficult to correct $\beta 2$ not repeating this kind of computation.

5. PARTIAL COMPUTATION AND LEAST FIXPOINT

Generalized partial computation is similar to a program transformation technique [9] often used to derive a least fixpoint of a recursive function. Let β be a strictly correct β partial evaluator, $\tau[f]$ be a functional, $f[u]==\tau[f][u]$ be a recursive program, and $lf\tau$ be a least fixpoint of $\tau[f]$. Then $lf\tau=lub\{\tau^n[\Omega]\}$. Let ***eval*** be a fixpoint interpreter. Then

$\beta[\tau[f][u];\phi]\equiv\tau[f][u]$ (from strict correctness of β)
$\tau[f][u]\equiv lf\tau[u]$ (from the definitions of ***eval*** and $lf\tau$)

Therefore $lf\tau[u]\equiv\beta[\tau[f][u];\phi]$. If the right side does not include a recursive call, $\lambda[[u];\beta[\tau[f][u];\phi]]$ is a fixpoint of $\tau[f]$. Since β often introduces recursive calls by use of partial definition, it is necessary to remove recursion from partial computation results to obtain least fixpoints.

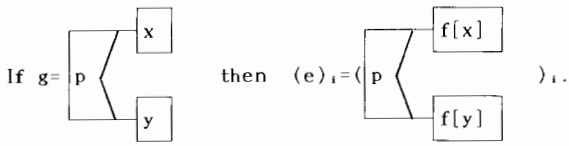
To derive a fixpoint of a recursive program, a program transformation rule called "distribution of a function over a conditional expression" or "merging of functions" is important. Partial computation method $\beta 3$ described below incorporates this rule (4.4) and two others (3 and 4.5) with $\beta 2$.

Example 10: Partial computation method $\beta 3$
Execute one of the four operations below depending on its effectiveness:

(1) If e is a conditional form then do the same as $\beta 2$.
(2) $(e)_i=e$.
(3) $(e)_i=(v)_i$ for u-form v such that $i\vdash^*\{v=e\}$.
(4) If e is a composite form such as $e=f[g]$ for a function f, then execute one of the following five operations depending on its effectiveness:

(4.1) Partially define $(e)_i$, i.e. $(e)_i=f_g{}'[u]$; $H=H\cup\{f_g{}'\};f_g{}'[u]<=e'$ where e' is one of the right side of the transformation rules of $\beta 3$ exept (4.1).

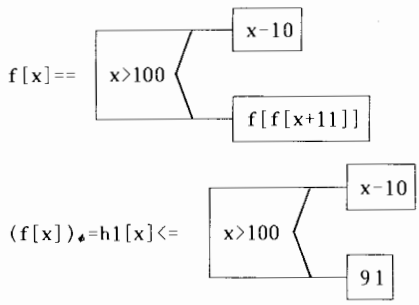
(4.2) If $(e)_i$ is partially defined, then let $f_d{}^j=\lambda[[u];m]$ be one of the partially defined functions (if a function with the closest partial information j to i is selected, the partial evaluator can be executed most quickly). Let g=d/k and $i \vdash^* j/k$, then $(e)_i=(f_d{}^j[k])_i$.
(4.3) If f is a nonprimitive function such as $f=\lambda[[u];b]$ then $(e)_i=(b/g)_i$.
(4.4) If g is a conditional form:



(4.5) $(e)_i=(f[\text{result of } (g)_i])_i$.

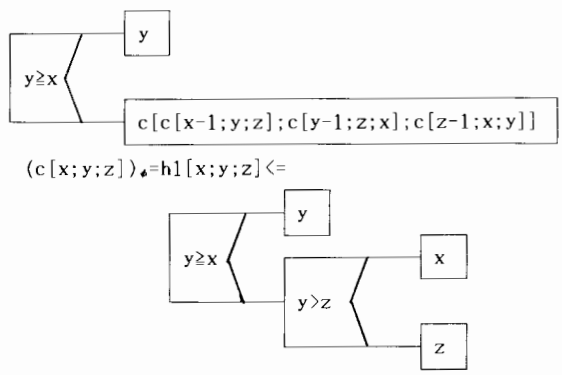
By combining $\beta 3$ with some recursion removal rules, least fixpoints of McCarthy's 91-function and Takeuchi's Tarai-function can be derived as follows:

Example 11: McCarthy's 91-function:



The derivation process is described in Appendix 3 which shows that $\beta 3$ is a highly nondeterministic procedure. To make $\beta 3$ more deterministic is a future research problem.

Example 12: Takeuchi's Tarai-function:



(Derivation process is omitted because of its lengthiness.)

6. CONCLUSION

The Generalized Partial Computation method and its applications to order changing program transformations have been presented in this paper. Proving the correctness of various partial computation procedures and their efficient implementation are future research problems. The idea of GPC came to one of the authors while visiting UPMAIL (Uppsala Programming Methodology and Artificial Intelligence Laboratory) from October, 1985 to September, 1986. The author is grateful to the members of UPMAIL for their fruitful discussion which encouraged him very much. The authors are also grateful to Academician A. P. Ershov for his long time encouraging them to continue partial computation research.

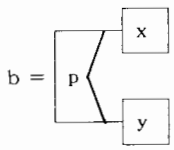
7. REFERENCES

1) Burstall, R.M. and Darlington, J.: A transformation system for developing recursive program, JACM, Vol.24, No.1, 1977, pp.44-67.
2) Futamura, Y.: Partial evaluation of computation process--an approach to a compiler-compiler, Computer, systems, controls 2, No.5, 1971, pp.45-50.
3) Futamura, Y.: Partial computation of programs, In E. Goto[et al](eds.),RIMS Symposia on Software Science and Engineering, Kyoto, Japan, 1982. Lecture Notes in Computer Science 147(1983) 1-35, Springer-Verlag.
4) Futamura, Y., Kawai, T., Tsutsumi, M. and Horikoshi, H.: Development of computer programs by Problem Analysis Diagram (PAD), Proc. of 5ICSE, IEEE Computer Society, New York, 1981.
5) Ershov, A.P.: Mixed computation in the class of recursive program schema, Acta Cybernetica, Tom.4, Fosc.1, Szeged, 1978.
6) Kahn, K. M.: A partial evaluation of Lisp written in Prolog, UPMAIL Report Department of Computing Science, Uppsala University, Uppsala, Sweden, March 11, 1982.
7) Kleene, S. C.: Introduction to Meta-Mathematics. North-Holland Publishing Co., Amsterdam, 1952.
8) Knuth, D. E., Morris, J. H. and Pratt, V. R.: Fast pattern matching in strings, SIAM Journal of Computer, Vol.6, No.2, June 1977, pp.323-350.
9) Manna, Z.: Mathematical Theory of Computation, McGRAW-HILL, 1974.
10) McCarthy, J. et al: LISP 1.5 Programmer's Manual M.I.T. Press Cambridge, Massachusetts, 1962.
11) Nelson, G. and Oppen, D. C.: Simplification by cooperating decision procedures, ACM TOPLAS, Vol.1, No.2, October 1979, pp245-257.
12) Turchin, V. F.: The concept of a supercompiler, ACM TOPLAS, Vol.8, No.3, July 1986, pp.292-325.

APPENDIX 1: DIFFERENCES BETWEEN α AND β

In this section, the basic idea of the conventional partial computation method α and its weakness are discussed. Partial computation of a simple pattern matcher concerning a given pattern is presented as an example. Let f= λ [[k;u];b]. The basic idea of α can be described by the following two steps (1) and (2) where ***eval*** is a fixpoint interpreter of recursive programs. α is named as an Interpreter Dependent Partial Computation because it uses an interpreter, ***eval***, when it evaluates conditions of a conditional expression.

Interpreter Dependent Partial Computation α :
(1) If b is a conditional expression, i.e.



then
(1.1) if eval[p;((k.k0))] is true, then $f_{k0}=\alpha[\lambda[[k;u];x];k0]$.
(1.2) if eval[p;((k.k0))] is false, then $f_{k0}=\alpha[\lambda[[k;u];y];k0]$.
(1.3) if eval[p;((k.k0))] is undefined, then

$f_{k0}=$ {p[k]}^{k0} ⟨ $\alpha[\lambda[[k;u];x];k0]$; $\alpha[\lambda[[k;u];y];k0]$ ⟩

where {p[k]}^{k0} is a form obtained by substituting k0 for all free occurrences of k in p.

(2) If b is not conditional, then $f_{k0}=\lambda[[u];\{b[k]\}^{k0}]$.

Syntactic differences between α and β are summarized in Table 1.

Table 1: Syntactic differences between α and β

	first argument	second argument	result
α	function f	known value k0 of variable k	f_{k0}
β	u-form e	predicate i on variable u	$(e)_i$

Differences between α and β in the power of partial computation are described below:

(1) β is more powerful than α in selecting one of two branches of a conditional expression. Assume that m>0 is known and and a[m;n] is Ackermann's function. If the value of m is unknown, eval[m>0;a] is undefined (where a is any environment not including m-value). Therefore, α cannot choose one of the conditional branches. On the contrary, since m>0⊢•m>0, β can choose the m>0 branch.

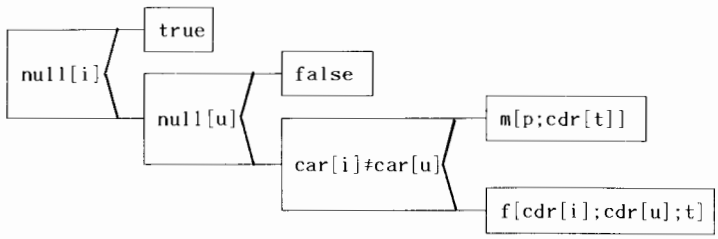
(2) Even when β cannot choose one of the conditional branches, it utilizes the condition, say p, of the branches, say x and y, in the later phase of partial computation. That is, $(x)_{i\wedge p}$ or $(y)_{i\wedge\neg p}$ will be performed. On the contrary, α does not use p in the partial computation of x and y. Therefore, β is considered more powerful than α.

To show the weakness of an Interpreter Dependent Partial Computation (IDPC), a simple pattern matcher is partially evaluated concerning a given pattern below. Note that the IDPC method being used here is an extension of α [3]. The extension is almost the same as the one used to obtain $\beta 2$ from β.

Let p and t be a pattern and a text, respectively. If p is contained in t, the value of a pattern matcher m[p;t] is true. If otherwise, the value is false. m is defined by using the auxiliary function f below. Symbol == is used to represent recursive definition of functions.

```
m[p;t]==f[p;t;t]

f[i;u;t]==
```

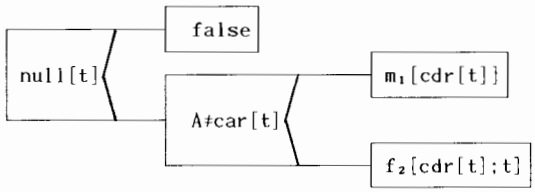


(where p is global).

The pattern matcher m will be partially evaluated concerning p= (A A A B). The result is represented as m_1. For notational convenience, lists are represented as follows:

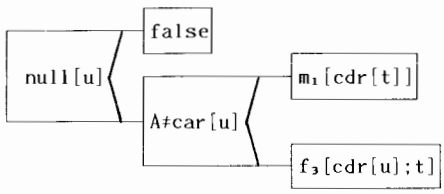
1 for (A A A B)
2 for (A A B)
3 for (A B)
4 for (B)
5 for ()

$m_1[t]==f_1[t;t]==$

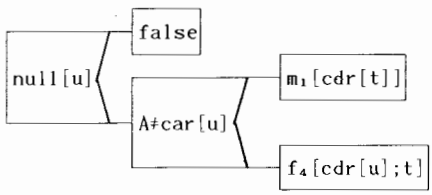


(from null[i]=null[(A A A B)]=false and car[i]=car[(A A A B)]=A)

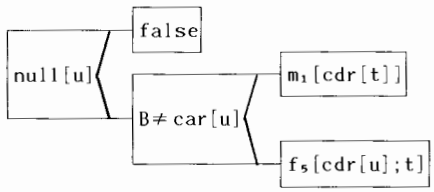
$f_2[u;t]==$



$f_3[u;t]==$



$f_4[u;t]==$



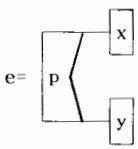
$f_5[u;t]==true$

The $m_1[t]$ is almost four times as large as m[p;t]. However, $m_1[t]$ is still an O(m*n) time program where m and n are the lengths of p and t, respectively. On the contrary, h1[t], obtained by using $\beta 2$ in Section 4, is an O(m+n) program.

APPENDIX 2: CORRECTNESS PROOF OF $\beta 1$

Two lemmas are given before the proof of Theorem 1.

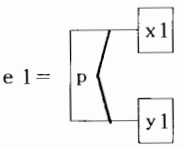
Lemma 1: Let i be any u-information and e be a conditional u-form such that



then (1) if $i \vdash^{\bullet} p$ then $e \subseteq^{i} x$.
(2) if $i \vdash^{\bullet} \neg p$ then $e \subseteq^{i} y$.

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

Lemma 2: Let e be a conditional u-form the same as above and $e1$ be a conditional u-form described below. Then $e \supseteq^{i} e1$ when $x \supseteq^{i \wedge p} x1$ and $y \supseteq^{i \wedge \neg p} y1$.



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

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, $\beta 1[e;i] \supseteq^{i} e$ will be proved.

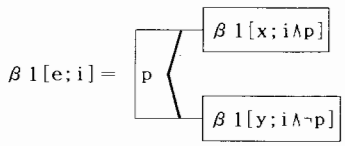
(1) When e is not a conditional form, $\beta 1[e;i]=e \supseteq^{i} e$.

(2) When e is a conditional form,

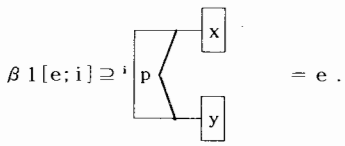
(2.1) Assume that $i \vdash^{\bullet} p$:
$\beta 1[e;i]=\beta 1[x;i] \supseteq^{i} x$ (from the induction hypothesis)
$\supseteq^{i} e$ (from Lemma 1).

(2.2) Assume that $i \vdash^{\bullet} \neg p$: The same as above.

(2.3) Assume that neither $i \vdash^{\bullet} p$ nor $i \vdash^{\bullet} \neg p$:



and $\beta 1[x;i\wedge p] \supseteq^{i\wedge p} x$ and $\beta 1[y;i\wedge\neg p] \supseteq^{i\wedge\neg p} y$ from the induction hypothesis. Therefore, from Lemma 2,



$= e$.

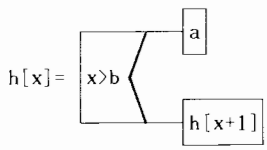
(QED)

Proof of Theorem 2: Replace $\supseteq^i$ by $\equiv^i$ in the proof above.

APPENDIX 3: PARTIAL COMPUTATION OF MCCARTHY'S 91-FUNCTION

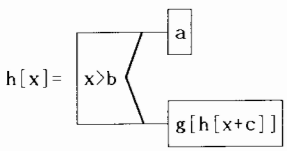
To obtain the minimal fixpoint of McCarthy's 91-function, the following two recursion removal rules are used where a, b and c stand for forms not containing x as a free variable:

(1)



and $h[x]=a$ for any x.

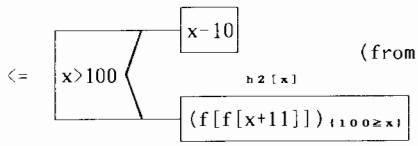
(2)



If $c>0$ and $g[a]=a$ then $h[x]=a$ for any x where g is a function not containing x as a free variable.

The correctness of the rules above is trivial.

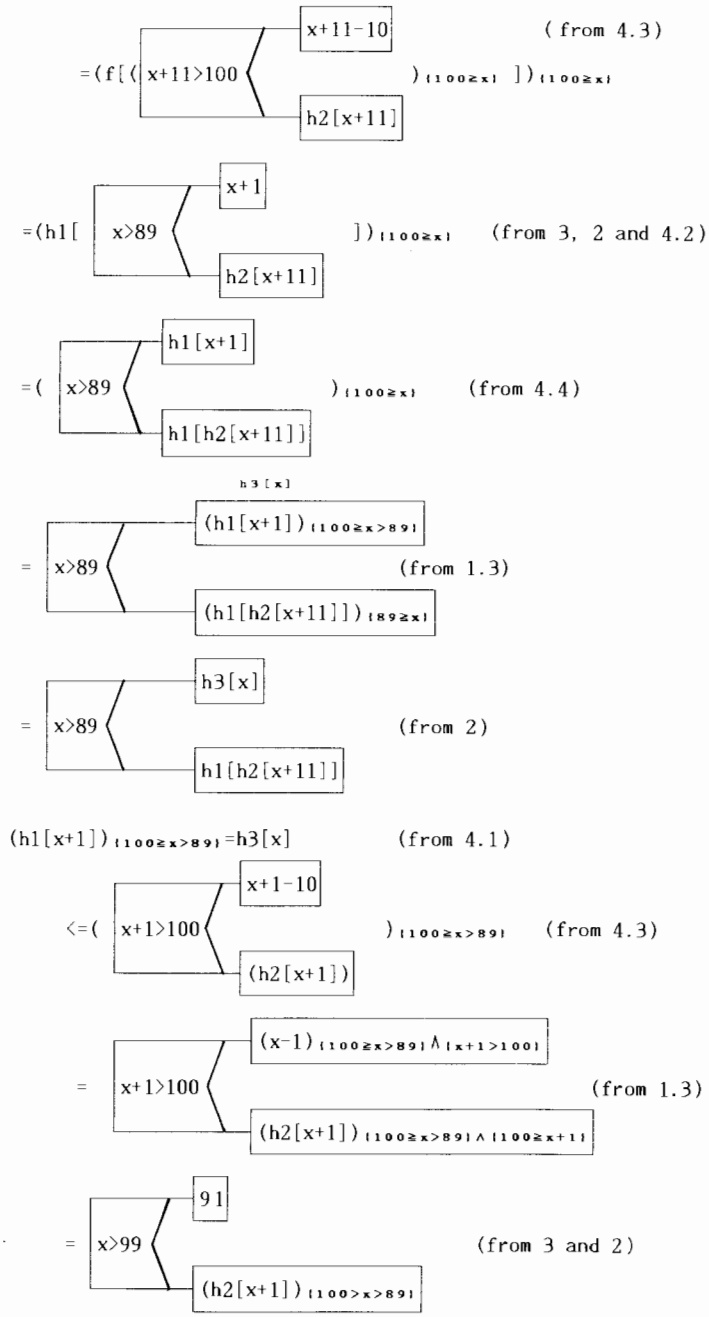
$(f[x])_\phi = h1[x]$ (from 4.1) Note that h1 stands for f_x^ϕ: The h2 and h3 below are used in a similar way.

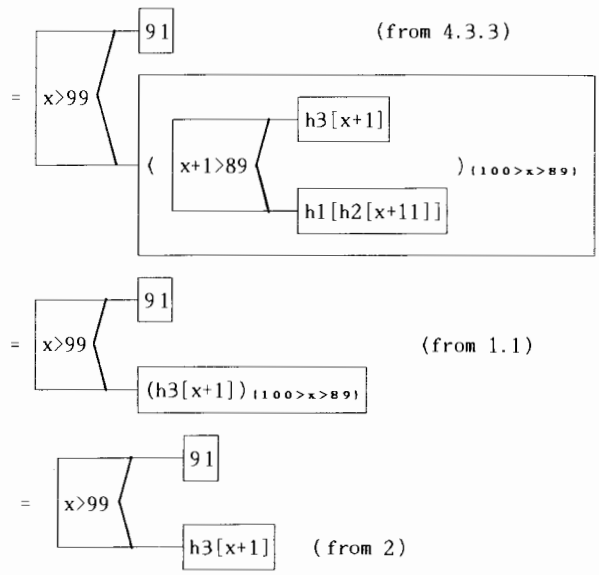


$(f[f[x+11]])_{(100\geq x)} = h2[x]$ (from 4.1)

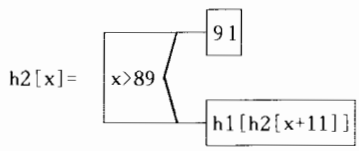
$\leq (f[(f[x+11])_{(100\geq x)}])_{(100\geq x)}$ (from 4.5)

$= (f[(h1[x+11])_{(100\geq x)}])_{(100\geq x)}$ (from 4.2)


=(f[(x+11>100 x+11-10 h2[x+11])100≧x])100≧x
(from 4.3)
=(h1[x>89 x+1 h2[x+11]])100≧x
(from 3, 2 and 4.2)
=(x>89 h1[x+1] h1[h2[x+11]])100≧x
(from 4.4)
h3[x]
= x>89 (h1[x+1])100≧x>89 (h1[h2[x+11]])89≧x
(from 1.3)
= x>89 h3[x] h1[h2[x+11]]
(from 2)
(h1[x+1])100≧x>89=h3[x]
(from 4.1)
<=(x+1>100 x+1-10 (h2[x+1]))100≧x>89
(from 4.3)
= x+1>100 (x-1)100≧x>89∧x+1>100 (h2[x+1])100≧x>89∧100≧x+1
(from 1.3)
= x>99 91 (h2[x+1])100>x>89
(from 3 and 2)



From the rule (1), h3[x]=91. Therefore,



Since h1[91]=91, h2[x]=91 from the rule (2). Therefore,