TM-0164

Partial Evaluation of Prolog Programs and its Application to Meta Programming

by
A. Takeuchi and K. Furukawa

September, 1986

PARTIAL EVALUATION OF PROLOG PROGRAMS AND ITS APPLICATION TO META PROGRAMMING

Akikazu Takeuchi and Koichi Furukawa
ICOT Research Center
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108 Japan

Partial evaluation is an important technique in computer science. In this paper, we present 1) the method of partial evaluation of Prolog programs, 2) its implementation in Prolog and 3) its application to meta programming. Meta programming plays an important role in Prolog application, because of its expressive power. However, efficiency has always been a problem in meta programming. We will show that the efficiency problem can be solved by specializing meta programs by partial evaluation with respect to object programs without losing the expressive power of meta programming. Furthermore we propose a new method for building inference systems. The new method is based on meta programming and utilizes partial evaluation as a basic tool. In a practical inference system, it is important that the system has the ability to acquire inference rules incrementally. We will show that our new method is also applicable to evolving systems by specifying the algorithm to specialize a meta interpreter incrementally with respect to an incrementally generated object program.

1. INTRODUCTION

Logic programming has provided powerful concepts for building expert systems, programming environments and database management systems. A common feature in these systems is that meta programs, especially meta interpreters, play an important role. In fact, it is quite natural to realize shells of expert systems and managers of database systems as meta interpreters specialized to these systems. The use of meta interpreters has the following advantages. 1) One can clearly separate object-level control and meta-level control. 2) Because of 1), the system is easy to understand and easy to modify. One problem, however, is that interpretive execution of object programs by the meta interpreter makes the system quite slow. We have developed the partial evaluation program, PEVAL, and have shown that the performance of the system can be improved by partially evaluating the meta interpreter with respect to the specific object program {1}. PEVAL has been extended to partially evaluate Prolog programs more flexibly.

Several researchers have studied the partial evaluation of Prolog programs. Komorowski investigated partial evaluation as a part of a theory of interactive, incremental programming {2}. Kahn developed the partial evaluator of Lisp programs in Prolog and succeeded in generating the compiler of his LM-Prolog {3}. Venken developed a partial evaluator of Prolog in Prolog and investigated its application to query optimization {4}. Gallagher investigated the transformation of logic programs by the partial evaluation of interpreters and applied it to efficient implementation of non-standard control strategies {5}.

In this paper, we will describe the method of partial evaluation of Prolog programs realized in PEVAL and explain the application of partial evaluation to the specialization of meta programs with respect to object programs. Based on the experiments on specialization of meta programs, we propose a new methodology for building inference systems which utilizes partial evaluation as a kernel tool. Furthermore, it will be shown that meta programs can be specialized incrementally with respect to an object program constructed incrementally. This incremental specialization matches incremental rule acquisition in inference systems and the open world assumption in programming environments.

2. PARTIAL EVALUATION OF PROLOG PROGRAMS

Partial evaluation of a program is generally done in order to specialize an original general program to a special efficient program using information about the run-time environment. The input data for a program is an important part of such information. In the case of Prolog programs, the following two kinds of information can be regarded as input data:

(1) The goal statement, especially values of arguments of goals
(2) A set of clauses used as input data

In partial evaluation of Prolog programs, it is desirable to be able to partially evaluate a program with respect to both kinds of input data. The basic principle of partial evaluation is to evaluate parts of a program which do have enough input data and to keep them as they are if the parts do not have enough data. Concrete values of variables in an expression are usually necessary for evaluation in value oriented languages like functional programming languages. To evaluate an expression without enough values, some special evaluation scheme such as lazy evaluation is required. In the case of Prolog, the basic computation is based on unification, thus no special evaluation scheme is required for partial evaluation. This is an important advantage of Prolog.

The computation of a Prolog program can be regarded as depth-first and left-to-right traversal of the AND-OR tree corresponding to the program. Our partial evaluator, called PEVAL, examines this tree using the given partial data, expands a goal by the body of a clause, the head of which can be unified with the goal, and cuts off branches known to fail. Several strategies are possible for traversing by the partial evaluator such as top-down, bottom-up and middle-out. PEVAL, like Prolog, uses top-down and left-to-right. PEVAL starts the partial evaluation from the definition of the top-level goal and goes down through those of its descendent goals. We call this kind of control of partial evaluation "*goal-directed* partial evaluation".

The basic algorithm for goal-directed partial evaluation is similar to that used in searching for all solutions for a given goal. In fact, if an original program has a finite AND-OR tree, the extreme case of partial evaluation involves searching for all instances of the top-level goal. For partial evaluation of a program with an infinite AND-OR tree there must be some mechanism to detect a loop of partial evaluation and deal with it. PEVAL detects a loop when the following condition is satisfied:

```
peval_goal(Goal,inf_loop,Stack) :-
     second_occurence(Goal,Stack),!.
peval_goal(Goal,NewDef,Stack):-
     clauses(Goal,Cls), !,
     head_unif(Cls,Goal,Selected),
     peval_clauses(Selected,NewDef,
                      [Goal|Stack]).

peval_clauses([],fail,_) :- !.
peval_clauses(Clauses,Ans,Stack) :-
     peval_clauses1(Clauses,Temp-[],Stack),
     close(Temp,Ans).

peval_clauses1([cl(Head,Done,[Goal|Rest])|
                  Cls],Ans,Stack) :- !,
     peval_goal(Goal,Subs,Stack),
     unfold(Subs,cl(Head,Done,[Goal|Rest]),
                 CCls-Cls),
     peval_clauses1(CCls,Ans,Stack).
peval_clauses1([cl(Head,Done,[])|Cls],
                 [cl(Head,Done,[])|NTail]-NT,
                 Stack) :- !,
     peval_clauses1(Cls,NTail-NT,Stack).
peval_clauses1([],T-T,_).

unfold(inf_loop,cl(Head,DH-[G|DT],[G|R]),
       [cl(Head,DH-DT,R)|Tail]-Tail) :- !.
unfold(fail,_,Tail-Tail) :- !.
unfold([],_,Tail-Tail) :- !.
unfold([cl(Head,Body,[])|Ss],
       Clause,[Nclause|T]-Tail) :-
     expand(Clause,cl(Head,Body,[]),Nclause),
     !, unfold(Ss,Clause,T-Tail).

expand(Clause,cl(G,B-[],[]),
       cl(Head,DH-NDT,Gs)) :-
     copy(Clause,cl(Head,DH-DT,[G|Gs])),
     append(B,NDT,DT).

head_unif([(H:-Body)|Cls],Goal,
          [cl(H,D-D,Blst)|Tail]):-
     copy(Goal,H), !,
     and_to_list(Body,Blst),
     head_unif(Cls,Goal,Tail).
head_unif([_|Cls],Goal,Tail) :-
     head_unif(Cls,Goal,Tail).
head_unif([],_,[]).

close([],fail) :- !.
close(X,X).
```

Fig. 1 Basic algorithm of PEVAL

Let G' be a goal appearing as a new goal to be partially evaluated during the partial evaluation of the clause defining a goal G.

A loop is detected if
(1) G' is a variant of G or
(2) G' is an instance of G

where a variant of a goal G is defined to be the same goal as G except the names of variables.

The first case is clearly a loop. An infinite sequence of goals where arguments of a goal are incrementally instantiated to infinite terms, *e. g.* {`p(X1)`, `p([a|X2])`, `p([a,a|X3])`, ...}, is detected as a loop by the second case. It is not necessary to handle a generalizing sequence of goals, *e. g.* {`p([a,a,a|X1])`, `p([a,a|X2])`, `p([a|X3])`, ...}, since it has the most general goal as a limit. The condition (2) is too strong, and may regard a safe computation as a loop. When a loop is detected, PEVAL stops further partial evaluation and returns a goal as the result of partial evaluation.

The basic algorithm of partial evaluation realized in PEVAL is shown in fig. 1. `peval_goal(Goal, NewDef, Stack)` is the relation which takes `Goal` and `Stack` as input and returns a specialized definition `NewDef` of the goal, where `Stack` is a stack holding goals being partially evaluated. The first clause specifies the case in which the loop detection condition above is satisfied. In this case, an atom, `inf_loop`, is returned instead of a specialized definition. In the second clause, clauses, that can be unified with the goal, are partially evaluated by `peval_clauses`. `peval_clauses(Clauses, NewClauses, Stack)` is the relation taking `Clauses` and `Stack` as input and returns the result, `NewClauses`, of partial evaluation of `Clauses`. The first clause of `peval_clauses` specifies the case in which no clause can be unified with the goal, in which an atom, `fail`, is returned. The second clause specifies the remaining case, in which a body of each candidate clause is partially evaluated by `peval_clauses1` and `fail` is returned if the result of `peval_clauses1` is empty, otherwise the result of `peval_clauses1` is returned as the result of `peval_clauses`. For each clause in the first argument, `peval_clauses1` partially evaluates goals in the body from left to right by `peval_goal` and expands the goal with the new definition by `unfold` and further partially evaluates the remaining goals in the expanded clauses by `peval_clauses1`.

In actual implementation, PEVAL is augmented at many points to enable user-control so that PEVAL can process as many kinds of programs as possible. PEVAL can handle the following types of programs.

- Programs which include arbitrary system predicates except *cut* (`if` statement is available instead of *cut*).
- Programs which are open, that is, which have undefined predicates.

Currently, PEVAL does not handle the *cut* operator since the cut is too primitive to handle. Relatively high control constructs such as `if` and `case` are easy for PEVAL to handle. Thus, instead of directly handling *cut* in PEVAL, a program using *cut* is first translated into a program using `if` and then it is partially evaluated.

PEVAL provides several control primitives for a user to instruct PEVAL to totally evaluate some parts of a program, to stop evaluation of other parts, to expand loops detected by the second condition if necessary, to inhibit unfolding of pre-specified predicates and so on. PEVAL can safely handle open programs with these controls, and the user can control the size of codes generated by PEVAL. A predicate with an incomplete (not empty) definition cannot be handled by PEVAL. The treatment of such predicates will be discussed in section 5. The new definitions generated by the partial evaluation are stored in the internal database. When a new clause is obtained, PEVAL checks whether the new clause is already in the database or not to ensure that the new program has no redundancy.

3. APPLICATION TO META PROGRAMMING

3.1 Meta programming

Meta programming is a widely used programming technique in logic programming. Meta programming can be described informally in the following way.

(1) To handle a program as data
(2) To handle data as a program and to evaluate it
(3) To handle a result (success or fail) of a computation as data

The best known example of meta programming is the *demo* predicate of Bowen and Kowalski {6}. *demo* takes two arguments, a program and a goal. Using this *demo* predicate, Bowen and Kowalski describe powerful programming examples amalgamating object and meta languages. Meta programming is also powerful in problem solving. Bundy has shown efficient meta-level control of problem solving {7}.

```
solve(true,[100]).
solve((A,B),Z) :-
    solve(A,X), solve(B,Y), append(X,Y,Z).
solve(not(A),[CF]) :-
    solve(A,[C]), C < 20, CF is 100-C.
solve(A,[CF]) :-
    rule(A,B,F) , solve(B,S), cf(F,S,CF).

cf(X,Y,Z) :-
    product(Y,100,YY), Z is (X*YY)/100.
product([],A,A).
product([X|Y],A,XX) :-
    B is X*A/100, product(Y,B,XX).

rule(A,B,F) :- ((A:-B)<>F).
rule(A,true,F) :- (A<>F).
```

(a) Meta Program

```
should_take(Person,Drug) :-
    complains_of(Person,Symptom),
    suppresses(Drug,Symptom),
    not(unsuitable(Drug,Person)) <> 70.

suppresses(aspirin,pain) <> 60.
suppresses(lomotil,diarrhoea) <> 65.

unsuitable(Drug,Person) :-
    aggravates(Drug,Condition),
    suffers_from(Person,Condition) <> 80.

aggravates(aspirin,peptic_ulcer) <> 70.
aggravates(lomotil,
           impaired_liver_function) <> 70.
```

(b) Object Program

Fig. 2 Meta Interpreter and Object Program

Meta programming is also important in building programming environments. The algorithmic program debugging system developed by Shapiro {8} uses meta programming extensively.

3.2 Partial evaluation of a meta program

It has been said that the meta programming is a useful approach because of its expressive power, however it is inefficient because of the layers of interpretations. We claim that *partial evaluation can solve the inefficiency problem of meta programming*. It is possible to translate a meta program into an efficient program by partial evaluation. In fact, since an object program can be regarded as input data from a meta program, the meta program can be specialized by partial evaluation with respect to the object program, so that the layers of interpretations disappears. This significantly improves the efficiency of the program and encourages users to utilize fully the expressive power of meta programming.

We demonstrate this claim on the Horn clause meta interpreter handling certainty factors originally deviced by {9}. Fig. 2 (a) shows the meta interpreter. The **solve** predicate is a binary relation, the first argument of which is a conjunction of goals to be solved and the second argument of which is the list of certainty factors of the goals when they are solved. Fig. 2 (b) shows an object program, which is a set of rules used to recommend a drug to a patient. "<> N" attached to each clause indicates the certainty factor of that clause (inference rule).

Fig. 3 shows the result of partial evaluation. In contrast with the original meta program which is general, the program in fig. 3 is specific to the object program (fig. 2 (b)). By comparing the resulting program with the original object program, the specialized program can be regarded as that obtained by augmenting the object program to handle certainty factors. Note that the resulting program has the same structure as the object program. It is possible

Table 1 Comparison of execution time

(CPU Time)

	p_1	p_2	p_3
Interpretive Execution	1674	1157	901
Compiled Execution	110	46	39

p_1: Meta + Object program (fig. 2)
p_2: The specialized program (fig. 3)
p_3: The specialized program (fig. 4)

to partially evaluate the resulting program further. Fig. 4 shows the new specialized program. In the resulting program, all the object goal invocations except **should_take**, **complains_of** and **suffers_from** are expanded. As a result, this program is a little more efficient than the program in fig. 3.

```
solve(should_take(A,B),[C]) :-
    solve(complains_of(A,D),E),
    solve(suppresses(B,D),F),
    solve(unsuitable(B,A),[G]),
    G<20,H is 100-G,append(F,[H],I),
    append(E,I,J),cf(70,J,C).
solve(suppresses(aspirin,pain),[60]).
solve(suppresses(lomotil,diarrhoea),[65]).
solve(unsuitable(A,B),[C]) :-
    solve(aggravates(A,D),E),
    solve(suffers_from(B,D),F),
    append(E,F,G),cf(80,G,C).
solve(aggravates(aspirin,
                 peptic_ulcer),[70]).
solve(aggravates(lomotil,
           impaired_liver_function),[70]).
```

Fig. 3 Result of Partial Evaluation

```
solve(should_take(A,aspirin),[B]) :-
    solve(complains_of(A,pain),C),
    solve(suffers_from(A,peptic_ulcer),D),
    cf(80,[70|D],E),E<20,F is 100-E,
    append(C,[60,F],G),cf(70,G,B).
solve(should_take(A,lomotil),[B]) :-
    solve(complains_of(A,diarrhoea),C),
    solve(suffers_from(A,
              impaired_liver_function),D),
    cf(80,[70|D],E),E<20,F is 100-E,
    append(C,[65,F],G),cf(70,G,B).
```

Fig. 4 Another Resultof Partial Evaluation

Table 1 shows the comparison of execution times of various programs.

4. PARTIAL EVALUATION AS A BASIC TOOL FOR BUILDING INFERENCE SYSTEMS

It has been said that Prolog is a good base language for building inference systems for the following reasons.

(1) **Unification:** Computation based on unification is more powerful than the pattern matching and pattern driven computation of conventional AI languages.
(2) **Backtracking search:** Prolog provides automatic backtracking search essential in an inference system

However, the fact that Prolog is a good base language does not suggest any special method for building inference systems. Many methods have been proposed so far. In the following we summarize these methods so far, clarify the problems and propose a new method using partial evaluation as a basic tool.

Conceptually, inference systems consist of two components: inference rules and an inference engine. Inference rules are domain specific knowledge. In contrast, the inference engines are domain independent and perform inferences using some strategy based on the inference rules. Meta-level control is implemented in the inference engine.

There are two typical methods for building an inference system in Prolog.

(1) To implement the inference engine as an interpreter of inference rules
(2) To translate inference rules into an executable program

APES {10} and the meta interpreter handling certainty factors of Shapiro are examples of system using the first method. In APES inference rules are represented by Horn clauses and the inference engine is implemented as a meta interpreter including a facility for tracing the history of inferences. Examples of systems using the second approach are the expert system developed by Clark *et al.* {11} and the production system of Tanaka {12}. Parsing systems such as DCG and BUP {13} are other examples of such systems in which inference rules are translated into Prolog programs and executed directly by the underlying Prolog processor.

As mentioned in section 3, the first method makes it quite easy to understand the details of inference and easy to modify the inference strategy, while its slow efficiency is a serious disadvantage. In contrast, the second method is very efficient since inference is performed by direct execution of the translated Prolog program, but it is difficult to understand both the translation program and the translated program. This is because the translation program includes a lot that are irrelevant to inference such as syntax analysis and I/O, and the translated program is too specific, making it difficult to understand the basic inference strategy.

Thus the two methods have the complementary advantages and disadvantages. This reflects the trade-off between the efficiency of the specialized programs and the generality of the interpretive approach. We propose a new approach amalgamating both methods overcoming the trade-off by partial evaluation. Our method incorporates the advantages while avoiding the disadvantages of both methods. The system is first constructed as a combination of an inference engine and inference rules. Then the inference engine is partially evaluated with respect to the inference rules, and the resulting specialized program is executed. In the first stage, the inference engine is described as a general meta interpreter, which makes it easy to understand and modify the engine, and easy to implement meta-level control in the engine. On the other hand, since inference is performed by direct execution of the specialized program, high efficiency can be achieved, as demonstrated in section 3. Partial evaluation plays a central role in this new method, combining efficient inference and the generality, understandability and maintainability of the inference engine.

Several experiments were performed to verify the new method. Takewaki *et al.* applied partial evaluation to their algebraic manipulation system implemented in Prolog using meta programming extensively {14}. In their system, rewriting rules for expressions are clusterized with respect to their application conditions and stored in object level. Strategic control rules which select an appropriate rule set for an expression are described in meta level. They reported that the specialized program runs on an average five times faster than the original program. We describe the experiment on BUP (Bottom-up Parsing) {13} here. BUP is a bottom-up parser of context free grammars (CFG) and can be regarded as an inference system. Usually a BUP is constructed by the second method, that is, CFG rules are translated into a Prolog program by the BUP translator and parsing is performed by direct execution of the translated program. As mentioned above, however, understandability and maintainability are problems in this method. We demonstrate that the new method is applicable to bottom-up parsing and will achieve high efficiency, understandability and maintainability simultaneously. Fig. 5 shows the bottom-up interpreter for CFG rules (fig. 5 (a)), CFG rules (fig. 5 (b)) and the corresponding program generated by the BUP translator (fig. 5 (c)). In contrast with the BUP translator, the bottom-up interpreter is quite compact and it is easy to understand its bottom-up strategy. It is also easy to modify the strategy. Fig. 6 shows the result of the partial evaluation. The specialized program has the same structure as the translated program. Compared to the translated program, the efficiency of the result of partial evaluation is in no way inferior to that of the translated program. This proves that our method provides a new general approach for building inference systems, in which understandability, maintainability and high efficiency are all achieved simultaneously.

5. INCREMENTAL SPECIALIZATION OF META INTERPRETER

A pure Prolog program consists of a set of Horn clauses. The structure of a program is quite flexible since no meaning is attached to the order of clauses and the scope of a variable is closed in a clause. Like rule-based programming languages, it is easy to construct a program incrementally. Incremental programming is important when one builds an inference system. It is generally difficult to extract all the expert knowledge at once. Systems are developed incrementally as expert knowledge is extracted incrementally. It is this aspect of expert systems that makes Prolog and other rule-based languages so suitable for their inference systems. In the previous section we proposed a new methodology for constructing inference systems. In this section we show that the key idea of the new methodology, that is, specialization of the meta interpreter, is also applicable to the case in which an object program is constructed incrementally. We first summarize the properties of PEVAL, following the theory of partial evaluation {15}. Then we derive the algorithm which specializes a meta interpreter incrementally with respect to an object program constructed incrementally.

Since PEVAL is written in Prolog, it is natural to represent it as a relation, *peval*.

$$peval(Program, Data, SpecializedProgram)$$

peval represents a relation which says that *SpecializedProgram* is a result of the partial evaluation of *Program* with respect to *Data*. Given a ternary relation $R(u, v, w)$, the result of partial evaluation of R with respect to the first argument u is denoted by $R_u(v, w)$. From the definition of partial evaluation, the following formulae hold.

$$R(u, v, w) \equiv R_u(v, w) \qquad (1)$$

$$peval(R, u, R_u). \qquad (2)$$

Representing the relation obtained by the partial evaluation of *peval* in the formula (2) with respect to some R by $peval_R(u, R_u)$, the following formulae are obtained as special cases of (1) and (2).

$$peval(R, u, R_u) \equiv peval_R(u, R_u)$$

$$peval_R(u, R_u). \qquad (3)$$

```
goal((P,Q),S0,S) :-
     goal(P,S0,S1), goal(Q,S1,S).
goal(C,S,S1) :-
     dict(F,S,S2),link(F,C),
     derive(F,S2,C,S1).
derive(F,S,F,S).
derive(F,S2,C,S1) :-
     rule1((Lemma <= (F,Rest))),
     link(Lemma,C), goal(Rest,S2,S3),
     derive(Lemma,S3,C,S1).
derive(F,S2,C,S1) :-
     rule2((Lemma <= F)),link(Lemma,C),
     derive(Lemma,S2,C,S1).
link(C,C).
link(F,C) :-
     rule1((Lemma <= (F,_))),link(Lemma,C).
link(F,C) :-
     rule2((Lemma <= F)),link(Lemma,C).
dict(F,[X|S],S) :- rule((F <= [X])).
rule1((A <= (B,C))) :- rule((A <= (B,C))).
rule2((A <= B)) :-
     rule((A <= B)),\+(B=(_,_)),
     \+(B=[_]).
```

(a) BUP interpreter

```
rule((s<=(np,vp))).    rule((np<=(det,n))).
rule((vp<=vi)).        rule((vp<=(vt,np))).
rule((n<=[boy])).      rule((n<=[girl])).
rule((vi<=[walks])).   rule((vt<=[likes])).
rule((det<=[a])).      rule((det<=[the])).
```

(b) CFG rules

```
dict(n,[],[boy|A],A).        link(X,X).
dict(n,[],[girl|A],A).       link(det,np).
dict(vi,[],[walks|A],A).     link(vt,vp).
dict(vt,[],[likes|A],A).     link(vi,vp).
dict(det,[],[a|A],A).        link(np,s).
dict(det,[],[the|A],A).      link(det,s).
vt(vt,X,Y,Y,X).    vi(vi,X,Y,Y,X).
det(det,X,Y,Y,X). n(n,X,Y,Y,X).
np(np,X,Y,Y,X).    vp(vp,X,Y,Y,X).
s(s,X,Y,Y,X).
np(B,[],C,D,E) :-
     link(s,B), goal(vp,[],C,F),
     call(s(B,[],F,D,E)).
det(B,[],C,D,E) :-
     link(np,B), goal(n,[],C,F),
     call(np(B,[],F,D,E)).
vi(B,[],C,D,E) :-
     link(vp,B), call(vp(B,[],C,D,E)).
vt(B,[],C,D,E) :-
     link(vp,B), goal(np,[],C,F),
     call(vp(B,[],F,D,E)).
goal(CurGoal,Arg,S0,S) :-
     dict(Nt,Arg1,S0,S1), link(Nt,CurGoal),
     functor(Pred,Nt,5),
     arg(1,Pred,CurGoal), arg(2,Pred,Arg1),
     arg(3,Pred,S1), arg(4,Pred,S),
     arg(5,Pred,Arg), call(Pred).
```

(c) Code generated by BUP translator

Fog. 5 Bottom-up Parser

Let us consider the relation I :

$$I(Program, Goal, Result)$$

I represents a meta interpreter taking two inputs, an object program *Program* and a goal *Goal*, and returning *true* to *Result* if *Goal* can be derived from *Program*, otherwise it returns *false*. By replacing R and u in formula (3) by I and a specific program P respectively,

$$peval_I(P, I_P). \qquad (4)$$

is obtained. In the theory of partial evaluation, $peval_I$ is known as the compiler corresponding to the interpreter I and I_P as the object code of P generated by the compiler $peval_I$.

```
dict(det,[a|A],A).          link(A,A).
dict(det,[the|A],A).        link(det,np).
dict(n,[boy|A],A).          link(det,s).
dict(n,[girl|A],A).         link(np,s).
dict(vi,[walks|A],A).       link(vi,vp).
dict(vt,[likes|A],A).       link(vt,vp).

derive(A,B,A,B).
derive(det,A,B,C) :-
     link(np,B), goal(n,A,D),
     derive(np,D,B,C).
derive(np,A,B,C) :-
     link(s,B), goal(vp,A,D),
     derive(s,D,B,C).
derive(vt,A,B,C) :-
     link(vp,B), goal(np,A,D),
     derive(vp,D,B,C).
derive(vi,A,B,C) :-
     link(vp,B), derive(vp,A,B,C).
goal((A,B),C,D) :-
     goal(A,C,E), goal(B,E,D).
goal(A,B,C) :-
     dict(D,B,E), link(D,A),
     derive(D,E,A,C).
```

Fig. 6 Partial Evaluation of BUP interpreter

Replacing R and u in the formula (3) by *peval* and I respectively, we have

$$peval_{peval}(I, peval_I). \qquad (5)$$

$peval_{peval}$ is known as a compiler-compiler which associates the interpreter I and its compiler $peval_I$. The derivation of (1) to (5) provides a brief overview of the theory of partial evaluation. In the following, we consider the method for incrementally specializing a meta interpreter with respect to an object program constructed incrementally.

Suppose that an object program P consists of n clauses, $P_1, ..., P_n$.

$$P = \bigwedge_{i=1}^{n} P_i$$

M and I^k are defined as follows.

M = the program of the meta interpreter I
I^k = the relation, whose program is $M + \bigwedge_{i=1}^{k} P_i$,

where $M + \bigwedge_{i=1}^{k} P_i$ denotes the program amalgamating M and $\bigwedge_{i=1}^{k} P_i$, and I^0 is defined to be I. I^k represents a partial system that can prove theorems derivable only from $\bigwedge_{i=1}^{k} P_i$.

By setting I and P in (4) to I^k and Q respectively,

$$peval_{I^k}(Q, I^k_Q) \qquad (6)$$

is obtained. Suppose that Q is equal to $\bigwedge_{i=k+1}^{n} P_i$. I^k_Q is the relation I^k specialized with respect to Q. In other words, I^k_Q is obtained by the partial evaluation of the program $M + \bigwedge_{i=1}^{k} P_i$ with $\bigwedge_{i=k+1}^{n} P_i$ added. Therefore, the following formula holds.

$$I^k_Q = I_P$$

Substituting for Q and I^k_Q in (6), we have

$$peval_{I^k}(\bigwedge_{i=k+1}^{n} P_i, I_P). \qquad (7)$$

$peval_{I^k}$ is the partially specialized compiler based on the interpreter I which, when the remaining program $\bigwedge_{i=k+1}^{n} P_i$ is given, generates the object code I_P of P. Let us consider the following formula:

$$peval(peval_{I^k}, P_{k+1}, S).$$

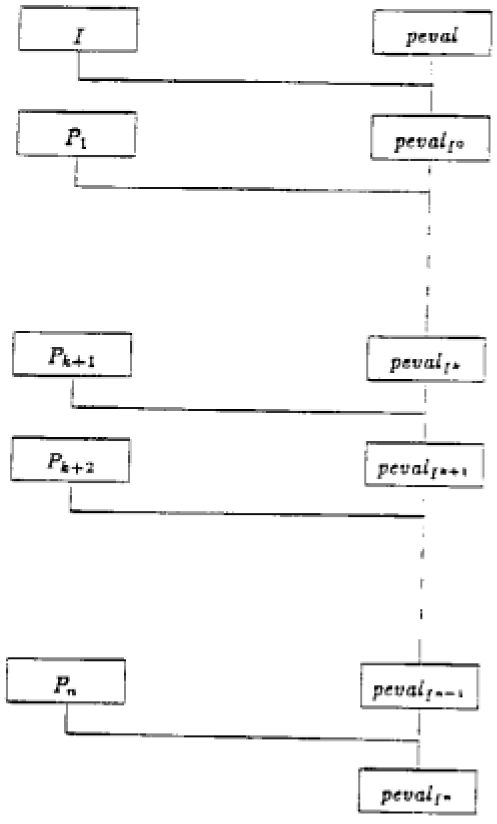

Fig. 7 Incremental Specialization

From the formula (7), given $\bigwedge_{i=k+2}^{n} P_i$, the relation S generates I_P. Thus, the relation S is nothing but $peval_{f^{k+1}}$, and

$$peval(peval_{f^k}, P_{k+1}, peval_{f^{k+1}}). \quad 0 \le k \le n-1 \qquad (8)$$

Partial evaluation of $peval_{f^k}$ with P_{k+1} given using this formula proves that $peval_{f^{k+1}}$ will be obtained. $peval_{f^k}$ is called the *partial compiler*. Formula (8) indicates the systematic method for generating the partial compiler (fig. 7). In this scheme, we can evolve a partial compiler as successive pieces of the object program are obtained. The partial evalaution of the interpreter with respect to P_k is done only once in the construction of $peval_{f^k}$ from $peval_{f^{k-1}}$, while it is done repeatedly in all the stages after the k-th stage if in each stage a new partial compiler is generated from scratch. This implies that our scheme can remarkably improve the computational complexity for generation of partial compilers by avoiding repeated computations. The partial compiler of the arbitrary stage can be used properly as a compiler. If a set of clauses is given to the partial compiler, it generates the object code of the conjunction of the set of clauses and the partial object program obtained up to that point.

6. SUMMARY

In this paper, the Prolog implementation of a partial evaluator for Prolog programs and its application to meta programming was described. Meta programming plays an important role in Prolog programming because of its expressive power. Partial evaluation will make meta programming more practical by improving the efficiency of meta programs. We also described a new methodology for building inference systems and established its validity on several examples. It was also shown that the new method will enable the incremental specialization of a meta interpreter with respect to an object program constructed incrementally. This feature matches the incremental acquisition of rules in the inference system.

ACKNOWLEDGEMENT

We wish to express our thanks to Kazuhiro Fuchi, Director of ICOT Research Center, who provided us with the opportunity to pursue this research in the Fifth Generation Computer Systems Project at ICOT. We would also like to thank Hiroyasu Kondo, Masaru Ohki, Hajime Kitakami and other ICOT researchers who participated in discussions.

REFERENCES

1 A. Takeuchi, K. Kondo, M. Ohki and K. Furukawa, An Application of Partial Computation to Meta Programming, *WG memo 85-SF-13*, Japan Information Processing Society, June, 1985, 9-16. (in Japanese)

2 J. Komorowski, *A Specification of Abstract Prolog Machine and Its Application to Partial Evaluation*, Linkoping Studies in Science and Technology Dissertations, no. 69, Software Systems Research Center, Linkoping University, 1981.

3 K. Kahn, *A Partial Evaluator of Lisp Written in a Prolog Written in Lisp Intended to be Applied to the Prolog and Itself which in turn is Intended to be Given to Itself Together with the Prolog to Produce a Prolog Compiler*, UPMAIL, Dept. of Computing Science, Uppsala University, 1982.

4 R. Venken, A Prolog Meta-Interpreter for Partial Evaluation and Its Application to Source to Source Transformation and Query-Optimisation, In *Proc. of ECAI-84*, North-Holland, 1984, 91-100.

5 J. Gallagher, *Transforming Logic Programs by Specialising Interpreters*, Dept. of Computer Science, Trinity College, University of Dublin, 1984.

6 K. Bowen and R. Kowalski, Amalgamating Language and Metalanguage in Logic Programming, In *Logic Programming*, K. Clark and S. Tarnlund (ed.), Academic Press, 1983, 153-172.

7 A. Bundy and B. Welham, Using Meta-level Inference for Selective Application of Multiple Rewrite Rules in Algebraic Manipulation, *Artificial Intelligence*, vol. 16, 1981, 189-212.

8 E. Shapiro, *Algorithmic Program Debugging*, The MIT Press, 1983.

9 E. Shapiro, Logic Programs with Uncertainties: A Tool for Implementing Rule-based Systems, In *Proc. IJCAI-83*, 1983, 529-532.

10 P. Hammond and M. Sergot, *apes: Augmented Prolog for Expert Systems*, Logic Based Systems Ltd. , 1984.

11 K. Clark and F. McCabe, Prolog: A Language for Implementing Expert Systems, In *Machine Intelligence*, D. Michie and Y. H. Pao (ed.), vol. 10, 1982.

12 H. Tanaka, Rule based Knowledge Representation in Prolog and its Application, IECE of Japan, *WG memo AL84-48*, 1984, 85-88. (in Japanese)

13 Y. Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi and H. Yasukawa, BUP: A Bottom-Up Parser Embedded in Prolog, *New Generation Computing*, vol. 1, no. 2, 1983, 145-158.

14 T. Takewaki, A. Takeuchi, S. Kunifuji and K. Furukawa, *Application of Partial Evaluation to the Algebraic Manipulation System and its Evaluation*, ICOT Tech. Resport TR-148, Institute for New Generation Computer Technology, 1985.

15 Y. Futamura, Partial Computation of Programs, *Journal of IECE of Japan*, vol. 66, no. 2, 1983, 157-165.