ICOT Technical Report: TR-126

TR-126

Partial Evaluation of Prolog Programs and its Application to Meta Programming

by
A. Takeuchi and K. Furukawa

July, 1985

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Partial Evaluation of Prolog programs and its Application to Meta Programming[1]

Akikazu Takeuchi and Kouichi Furukawa

ICOT Research Center
Institute for New Generation Computer Technology
Tokyo, Japan

ABSTRACT

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 progrgamming. Meta programming becomes to play very important role in Prolog application, because of its expressive power. However the efficiency was always problem of 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 loss of expressive power of meta programming. Furthermore we will 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 system has the ability to acquire inference rules incrementally. We will show that our new method is also applicable to these evolving systems by specifying the way to specialize meta interpreter incrementally with respect to an incrementally generated object program.

[1] The origirnal version of this paper was presented in Japanese at the Logic Programming Conference'85 (Tokyo) on July 2nd, 1985.

1 INTRODUCTION

Logic programming has provided powerful concepts for building expert system, programming environment and database management system. The features common in these systems are that meta programs, especially meta interpreters, play 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 approach by meta interpreters has the following advantages. 1) One can clearly separate object-level control and meta-level control. 2) Because of clear separation of object and meta programs, one can easily understand the system and can easily modify the system.

However, in the approach by meta interpreters, the performance of the system is quite low because of the interpretive execution of an object program by the meta interpreter. We have developed the partial evaluation program, PEVAL, and have shown that the performance of the system can be improved by partial-evaluating the meta interpreter with respect the specific object program (Takeuchi85).

PEVAL was extended to partial-evaluate Prolog programs more flexibly. In this paper, we will describe the method of partial evaluation of Prolog programs which is realized in the PEVAL and will explain the application of partial evaluation to the specialization of meta programs with respect to object programs. Based on the experiments of specialization of meta programs, we will propose a new method 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 open world assumption in programming environment.

2 PARTIAL EVALUATION OF PROLOG PROGRAMS

Generally partial evaluation of a program is to specialize an original general program to a special efficient program using information about run-time environment. The input data for a program is important one among such information. In the case of Prolog programs, the following two can be regarded as input data:

(1) Goal statement, especially values of arguments of goals

(2) A set of clauses which is used as input data

In partial evaluation of Prolog programs, it is desirable to be able to partial-evaluate a program based on both kinds of input data. In this chapter, we describe the method of partial evaluation of Prolog programs and explain the basic behavior of PEVAL which realizes the method. The range of Prolog programs which can be treated by PEVAL and the ways to control PEVAL are also shown.

The computation of a Prolog program can be regarded as traversal of the AND-OR tree corresponding to the program by depth-first and left-to-right strategy. Our partial evaluator, called PEVAL, examines this tree based on 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 which can be known to fail. For the traversal by partial evaluator, several strategies are known, such as top-down, bottom-up and middle-out. PEVAL uses top-down and left-to-right strategy as well as Prolog. Namely, PEVAL starts the partial evaluation from the definition of the top-level goal

and goes through those of its descendent goals. We call this kinds of control of partial evaluation goal directed partial evaluation.

The basic principle of partial evaluation is to evaluate parts of a program which have enough input data and to keep as it is if the parts don't have enough data. Generally, in the value oriented languages like functional programming languages, concrete values of variables in an expression are necessary for evaluation. In order to evaluate an expression without enough values, some special evaluation scheme such as lazy evaluation is required. However, in the case of Prolog, the basic computation is based on the unification, thus no special evaluation scheme is required for the partial evaluation. This is a very important feature of Prolog.

Keeping above observation in mind, let us partial-evaluate the following simple Prolog program.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parant(X,Z), ancestor(Z,Y).
```

where the following two clauses are given as an input clause set.

```
parent(taro,jiro).
parent(jiro,saburo).
```

Let us start partial evaluation of the above program from the goal `ancestor(X,Y)`. As a result of the partial evaluation, it is expected that the following specialized program will be obtained.

```
ancestor(taro,jiro).
ancestor(taro,saburo).
ancestor(jiro,saburo).
```

Note that the expected result of the partial evaluation coincides with the solution set of the goal `ancestor(X,Y)`. It means that the goal directed partial evaluation and searching for all solutions have the similar basic algorithm. In fact, if an original program has a finit AND-OR tree, extreme case of partial evaluation is nothing but searching for all instances of the goal literal.

For partial evaluation of a program with an infinit AND-OR tree, however, there must be some mechanism which detects a loop of partial evaluation and takes care of it. PEVAL detects a loop when the following condition is satisfied:

> *During the partial evaluation of the clauses defining a goal* `G`*, a goal* `G'`*, which is same as the goal* `G` *except the names of variables or is an instance of the goal* `G`*, appears as a new goal to be partial-evaluated.*

When a loop is detected, PEVAL stops further partial evaluation and returns a goal itself as a result of partial evaluation.

The basic algorithm of partial evaluation realized in PEVAL is shown in Figure 1. In Figure 1, peval_goal(Goal,NewDef,Stack) is the relation which takes a goal Goal, a stack Stack as input and returns a specialized definition of the goal NewDef, where Stack is a stack which keeps goals being partial-evaluated. The first clause specifies the case in which the condition of loop detection above is satisfied. In this case, an atom, inf_loop, is returned in the second argument. In the second clause, clauses, which define the goal and can be unified with the goal, are partial-evaluated by peval_clauses. peval_clauses(Clauses,NewClasues,Stack) is the relation which takes Clauses and Stack as input and returns NewClauses, where NewClauses is the result 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 this case, an atom, fail, is returned. The second clause specifies the remaining case, in which each body of each candidate clause is partial-evaluated by peval_clauses1 and fail is returned if the result of peval_clauses1 is

```
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).
```

Figure 1. Basic algorithm of PEVAL

empty, otherwise the result of `peval_clauses1` is returned as the result of `peval_clauses`. For each clause in the first argument, `peval_clauses1` partial-evaluates goals in the body from left to right by `peval_goal` and expands the goals with the new definition by `unfold` and further partial-evaluates the rest of goals in the expanded clauses by `peval_clauses1`.

Real PEVAL is augumented in many points in order to enable user control so that PEVAL can process as many kinds of programs as possible. PEVAL can handle the following kinds of programs.

- A program which includes arbitrary system predicates except *cut* (however, `if` statement is available).
- A program which is open, that is, which has undefined predicates.

Currently PEVAL does not treat *cut* operator since *cut* operator is too low primitive to handle. Relatively high control primitives such as `if` and case are easy to handle in PEVAL. Therefore, instead of handling directly *cut* in PEVAL, we adopt the approach in which a program using *cut* is first translated to a program using `if` and then it is partial-evaluated.

An user can control PEVAL by the following Horn clause forms:

(a) `type(Goal,Type) :- <Condition>.`

(b) `inhibit_unfolding(Goal) :- <Condition>.`

(c) `:- pmode <Mode declaration>.`

(d) `expand_loop(Goal) :- <Condition>.`

The form (a) says: "When a Goal `Goal` satisfies the condition `Condition`, then the type of evaluation of the goal is `Type`." There are three types.

- `e` ..... evaluable
- `t` ..... terminate
- `g` ..... partial evaluate

Type "e" means that the goal can be totally evaluable and type "t" indicates termination of partial evaluation of the goal. Type "g" is a default type which indicates the partial evaluation of the goal. Type "t" is useful when a program is open, that is, the program has undefined predicates. In such cases, an user can set the types of undefined predicates "t" in order to avoid the partial evaluation of undefined predicates. The form (c) specifies the mode of the predicate in the same way as DEC-10 Prolog. When PEVAL partial-evaluates a predicate, it first checks the mode declaration. If the mode is declared and at least one of the arguments specified as input mode is undefined, then PEVAL stops partial evaluation because there is not enough information. The form (b) controls the unfolding part of partial evaluation. When PEVAL gets the new definition of a predicate by partial evaluation, PEVAL usually unfolds the goal by the new definition. However, if the predicate satisfies the `inhibit_unfolding` relation then the unfolding of the goal by the new definition is inhibited and the new definition is stored in the internal database. This form is useful for reducing the amount of codes generated by partial evaluation. For example, suppose that, by the partial evaluation of p$\wedge$q, n and m clauses have been obtained as the new definitions of `p` and `q` respectively. If both `p` and `q` are unfolded by the new definitions, then $n \times m$ goal statements are generated. However, if neither goal are unfolded, then the amount of codes is $n + m$ clauses. The form (d) overrides the loop detection

condition described before. In PEVAL, when a new goal to be partial-evaluated is equal to the goal in the stack except the name of variables or the new goal is an instance of the goal in the stack, then the loop is detected. However if a new goal statisfies this form, then the second condition of the loop detection is ignored.

The new definitions generated by the partial evaluation are stored in the internal database. It may happen that the resulting program will have redundant clauses. Therefore, when a new clause is obtained, PEVAL checks whether the new clause is already in the database or not. An user must care about the definitions of the predicate, the partial evaluation of which is terminated by some reason, since the resulting program may become incomplete with respect to the definitions of such predicates. In the following cases, the resulting program may become incomplete.

(1) When unfoldings of some predicates are inhibited by the `inhibit_unfolding` predicate.

(2) When the types of some predicates are specified as "t" by the type predicate.

(3) When the partial evaluations of some predicates are terminated by the loop detection.

In the case (1), PEVAL automatically stores the definitions of such predicates in the internal database. In the case (2), PEVAL stores warning saying that no definition is stored for such predicates in the internal database. Currently there is no mechanism which covers the case (3).

3 APPLICATION TO META PROGRAMMING

3.1 Meta programming

Meta programming is widely used programming technique in logic programming. Meta progamming can be featured 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 program as data

The most well known example of meta programming is the *demo* predicate of Bowen and Kowalski (Bowen83). *demo* predicate takes two arguments, a program and a goal. *demo* predicate becomes true if the goal can be derived from the program, otherwise *demo* fails.

demo(Program,Goal)
→ true if Program ⊢Goal
→ fail otherwise

The logical meaning of the *demo* predicate is the derivability of the goal from the given program. Using this *demo* predicate, Bowen and Kowalski have shown the powerful progamming examples which amalgate object and meta languages. Meta programming is also powerful in the problem solving. Bundy have shown the meta-level control of problem solving (Bundy81). Furthermore, meta programming is also important in building programming environment. The algorithmic program debugging system developed by Shapiro (Shapiro83a) uses meta programming very much.

3.2 Partial evaluation of a meta program

Most of meta programs written so far are variations of meta interpreter. For example, the debugger above is one of the variation of meta interpreter. APES (Hammond83) is a tool for building logic-based expert system, which utilizes meta programming. In APES, Horn clauses are regarded as inference rules and inference engine is realized as meta interpreter, in which explanation facilities are implemented. It is possible to handle certainty factors in this meta interpreter. Shapiro has proposed such meta interpreter that handles certainty factor (Shapiro83b).

It have been said that meta programming approach is good because of its expressive power, however it is inefficient. We claim that *partial evaluation can solve the inefficiency problem of meta programs.* It is possible to translate a meta program to 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 the partial evaluation if the object program is given, so that the specialized program have no interpretive code. Consequently the efficiency of the program will be improved. In this way, partial evaluation solves the inefficiency problem of meta programming and encourages users to fully utilize the expressive power of meta programming.

We explain it by example. In Figure 2 (a), Horn clause meta interpreter which handles certainty

---

```
solve(true,[100]).
solve((A,B),Z) :- solve(A,X), solve(B,Y), ap(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

Figure 2. Meta Interpreter and Object Progam

factor is shown. In the figure, solve predicate is a binary relation, the first argument of which is a goal to be solved and the second argument is the certainty factor of the goal when it is solved. In the figure 2 (b), an object program is shown, which is a set of rules used to recommend a drug to a patient. "<> N" attached for each clause indicates certainty factor of that clause (inference rule).

In Figure 3 , instructions to PEVAL specified by an user and the resulting program are shown. It proves that the resulting program is the meta program which is specialized to the object

---

```
type(solve(complains_of(_,_),_),t) :- !.
type(solve(suffers_from(_,_),_),t) :- !.
type(solve(_,_),g) :- !.
type(rule(_,_,_),e) :- !.
type(cf(X,Y,Z),e) :- integer(X), ground(Y), !.
type(cf(X,Y,Z),t) :- var(X) ; \+(ground(Y)).
type(ap(X,Y,Z),e) :- fixed_length(X),!.
type(ap(X,Y,Z),t) :- \+(fixed_length(X)),!.
type(X<Y,e) :- integer(X),integer(Y),!.
type(X<Y,t) :- var(X) ; var(Y).
type(X>Y,e) :- integer(X),integer(Y),!.
type(X>Y,t) :- var(X) ; var(Y).
type(X is Y,e) :- ground(Y),!.
type(X is Y,t) :- \+(ground(Y)),!.
inhibit_unfolding(solve(Goal,_)) :-
   \+(Goal=true),\+(Goal=(P,Q)),
   \+(Goal=not(P)).
```

(a) Instruction to PEVAL

```
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,ap(F,[H],I),ap(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),
   ap(E,F,G),cf(80,G,C).
solve(aggravates(aspirin,peptic_ulcer),[70]).
solve(aggravates(lomotil,impaired_liver_function),[70]).

solve/2 unprocessed.
ap/3 unprocessed.
cf/3 unprocessed.
```

(b) Result of Partial Evaluation

Figure 3. Result of Partial Evaluation

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 (b))
p_3: The specialized program (Fig.4)

program and can not be used for other programs, while the original meta program can interpret any object program. By comparing the resulting program with the original object program (Fig.2 (b)), it proves that the specialized program can be seen as the variation of the object program which is augumented to handle certainty factors.

Note that the resulting program has the same structure as the object program. It is possible to partial-evaluate the resulting program more. If `inhibit_unfolding` is removed from the instructions to PEVAL (Fig.3 (a)), more partial-evaluated program will be obtained. In Figure 4, the new specialized program is shown. In the resulting program, all the object goal invocation except `should_take`, `complains_of` and `suffers_from` are expanded. Consequently this program will be a little more efficient than the program in Fig.3 (b).

In Table 1, the comparison of execution time of various programs is shown.

4 PARTIAL EVALUATION AS A BASIC TOOL FOR BUILDING INFERENCE SYSTEM

In this chapter, we investigate the implication of partial evaluation of meta programs from the point of view of building inference system.

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

(1) **Unification:** Computation based on unification is more powerful than pattern matching and

```
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,ap(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,ap(C,[65,F],G),cf(70,G,B).

solve/2 unprocessed.
cf/3 unprocessed.
ap/3 unprocessed.
```

Figure 4. Another Result of Partial Evaluation

pattern driven computation of conventional AI languages.

(2) **Search by backtracking:** Prolog provides automatic backtracking for search problem which is essential in inference system

However, the fact that Prolog is a good base language does not mean any special method for building inference system. Many methods have been proposed for building inference system so far. In the following we summarize the methods proposed fo far, clarify the problem and will propose a new method which uses partial evaluation as a basic tool.

Generally inference system consists of the following things.

(a) Inference Rules

(b) Inference Engine

Inference rules are domain specific knowledge. On the contrary, inference engine is domain independent and infers based on some strategy using the inference rules when a goal is given. Usually meta-level control is realized in inference engine.

Let us see examples. Production system is well known as rule-based inference system, in which production rules correspond to inference rules and engine of prodduction system corresponds to inference engine. Forward and backward production systems correspond to the inference systems which adopt forward (top-down) and backward (bottom-up) strategy respectively. Parsing system can be also regarded as inference system. In this case, grammatical rules and dictionary correspond to inference rules and parser corresponds to inference engine. In the case of parsing there are several algorithms such as top-down and bottom-up, which reflect the strategies adopted in the inference engine. Furthermore, Prolog interpreter can be also regarded as inference system, in which a Prolog program corresponds to inference rules and the interpreter corresponds to inference engine.

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

(1) To realize inference engine as an interpreter of inference rules

(2) To translate inference rules to an executable program

As examples of systems taking the first method, there are APES and meta interpreter handling certainty factor of Shapiro. In APES inference rules are represented by Horn clauses and inference engine is implemented as the meta interpreter in which the facility to collect the history of inference is implemented. Examples of systems taking the second method are the expert system by Clark et al.(Clark82) and the production system by Tanaka (Tanaka84). DCG and BUP (Matsumoto83) are another examples of this kind of systems. In these system, inference rules are translated to Prolog program and it is executed directly by the underlying Prolog processor.

Advantages of the first method are that it is quite easy to understand behavior of inference and is easy to modify inference strategy. Disadvantage of the first method is low efficiency. On the other hand, advantage of the second method is the high efficiency of inference since inference is performed by the direct execution of the translated Prolog program. Disadvantages are that it is difficult to understand both the translation program and the translated program, since the translation program includes many parts which is irrelevant to inference such as syntax analysis, I/O, and translated program is too specific to understand the basic inference strategy.

In this way, advantages and disadvantages of both methods are complementary. This complemental relation reflects the trade-off between efficiency of specialized program and generality of interpretive approach. We propose a new method which amalgamates both methods. The new method overcomes the trade-off by partial evaluation, so that our method have advantages of both method and have no disadvantages of either methods. In our method, the system is first built as a pair of inference engine and inference rules, second the inference engine is partial-evaluated with respect to the inference rules and then the resulting specialized program is executed. In the first stage, inference engine is described as general meta interpreter, so that it is easy to understand and modify the engine and is also 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 will be achieved. In this new method, partial evaluation plays a central role which combines the generality, understandability and maintenability of inference engine and efficient inference.

Several experiments were performed in order to verify the validity of the new method. Meta interpreter handling certainty factors presented in the chapter 3 is one of the experiments. In that example, we found that generality, understandability and maintenability are achieved by the clear separation of the meta interpreter (inference engine) and the object program (inference rules), and the efficiency is improved by the partial evaluation. As an another example verifying the validity of the new method, we show the experiment concerning BUP (Bottom-up Parsing) (Matsumoto83). BUP is a bottom-up parser of context free grammar (CFG) and can be regarded as inference system. Usually BUP is built by the method (2), that is, CFG rules are translated to Prolog program by the BUP translater and the parsing is performed by direct execution of the translated program. As mentioned above, however, understandability and maintenability of this method are the problem. We demonstrate that the new method can be also applicable to the bottom-up parsing and it will achieve the high efficiency, understandability and maintenability

simultaneously. In Figure 5, the bottom-up interpreter of CFG rules (Fig.5 (a)), CFG rules (Fig.5 (b)) and the corresponding program generated by the BUP translater (Fig.5 (c)) are shown. Comparing with the BUP translater, the bottom-up interperter is quite compact and

```
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).  dict(n,[],[girl|A],A).  dict(vi,[],[walks|A],A).
dict(vt,[],[likes|A],A).  dict(det,[],[a|A],A).  dict(det,[],[the|A],A).
link(X,X).  link(det,np).  link(vt,vp).  link(vi,vp).
link(np,s).  link(det,s).
vt(vt,_421,_422,_422,_421).  vi(vi,_443,_444,_444,_443).
det(det,_465,_466,_466,_465).  n(n,_487,_488,_488,_487).
np(np,_509,_510,_510,_509).  vp(vp,_531,_532,_532,_531).
s(s,_553,_554,_554,_553).
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

Figure 5. Bottom-up Parser

easy to understand its bottom-up strategy. It is also easy to modify the strategy. In Figure 6, the instruction used in the partial evaluation of the interpreter with respect to the given CFG rules and the result of the partial evaluation are shown. The specialized program has the same structure as the translated program. As compared with the translated program, the efficiency of the result of the partial evaluation is in no way inferior to that of the translated program. Consequently it proves that our new method provides a new approach for building inference system, in which understandability, maintenability and high efficiency are all achieved simultaneously.

5 INCREMENTAL SPECIALIZATION OF META INTERPRETER

In this chapter, following the theory of partial evaluation (Futamura83), the properties satisfied by the PEVAL is examined. Then we will derive the method which specializes a meta interpreter incrementally with respect to an object program which is constructed incrementally.

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

peval(*Program*, *Data*, *SpecializedProgram*)

```
type(dict(_,_,_),e).  type(rule1(_),e).
type(rule2(_),e).  type(link(_,_),e).
type(goal(_,_,_),g).  type(derive(_,_,_,_),g).

inhibit_unfolding(dict(_,_,_)).
inhibit_unfolding(link(_,_)).
inhibit_unfolding(goal(C,_,_)) :- \+(C=(P,Q)).
inhibit_unfolding(derive(_,_,_,_)).
```

(a) Instructions to PEVAL

```
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).
```

(b) Code generated by PEVAL

Figure 6. Partial Evaluation of BUP interpreter

peval represents a relation which says that $SpecializedProgram$ is a result of 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 formula holds.

$$R(u, v, w) \equiv R_u(v, w) \tag{1}$$

And also from the definition of *peval*, the following relation holds.

$$peval(R, u, R_u). \tag{2}$$

By representing the relation, which is obtained by the partial evaluation of *peval* in the formula (2) with respect to some R, by $peval_R(u, R_u)$, the following formula is obtained as a special case of the formula (1).

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

Therefore the following formula holds.

$$peval_R(u, R_u). \tag{3}$$

$peval_R$ is the specialized relation which is obtained by partial-evaluating *peval* with the first argument $Program$ fixed to a specific R. $peval_R$ represents the partial evaluator specialized to R. Let us consider the relation I :

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

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

$$peval_I(P, I_P). \tag{4}$$

is obtained. In the theory of partial evaluation, $peval_I$ is known as the compiler corresponding to the interpreter I and I_P as an object code of P. Thus $peval_I$ represents the relation expressing that I_P is an object code of P. From the formula (1),

$$I(P, G, R) \equiv I_P(G, R)$$

is obtained. This expresses that the execution of P in I is equivalent to direct execution of I_P.

By setting R and u to *peval* and I in the formula (3),

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

is obtained. $peval_{peval}$ is known as a compiler-compiler which associates the interpreter I and its compiler $peval_I$. These are the overview of the theory of partial evaluation. In the following, we consider the method to incrementally specialize a meta interpreter with respect to an object program which is constructed incremantally.

A pure Prolog program consists of a set of Horn clauses. The structure of a program is quite flexible since there is no meaning in the order of clauses and a scope of a variable is closed in a clause. Therefore, like rule-based programming languages, it is easy to construct a program incremantally. Incremental progamming is important when one will construct an expert system. Generally it is difficult to extract all the expert knowledge at once. Therefore usually the system is developed incrementally as expert knowledge are extracted incrementally . In this sense, Prolog and other rule-based languages are suitable for building expert system. In the previous chapter we have proposed a new method for building inference system. In this chapter we show that the key idea of the new method, that is, specialization of the meta interpreter, is also applicable to the case in which an object program will be constructed incrementally.

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

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

M and I^k are defined as follows.
M = the program of the meta interpreter I
I^k = the relation, the program of which is $M + \bigcap_{i=1}^{k} P_i$

where $M + \bigcap_{i=1}^{k} P_i$ denotes the program which amalgamates M and $\bigcap_{i=1}^{k} P_i$, and I^0 is defined as I. I^k represents a partial system which can prove theories which are derivable only from $\bigcap_{i=1}^{k} P_i$.

By setting I in the formula (5) to I^k,

$$peval_{peval}(I^k, peval_{Ik}). \qquad 0 \leq k \leq n \tag{6}$$

is obtained. In the case of $k = 0$, the formula is equivalent to (5). The formula represents the trivial solution to the incremental specialization, that is, the compiler $peval_{Ik}$ of the partial system I^k is generated from I^k by $peval_{peval}$. However, it may be difficult to make $peval_{peval}$. More feasible solution can be obtained from the following consideration.

By setting I and P to I^k and Q respectively,

$$peval_{Ik}(Q, I^k_Q). \tag{7}$$

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

$$I_Q^k = I_P$$

Therefore the following formula is obtained from (7).

$$peval_{I^k}(\bigcap_{i=k+1}^{n} P_i, I_P). \tag{8}$$

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

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

From the formula (8), given $\bigcap_{i=k+2}^{n} P_i$, the relation S generates I_P. Therefore the relation S is nothing but $peval_{IP_{k+1}}$.

$$peval(peval_{I^k}, P_{k+1}, peval_{I^{k+1}}). \qquad 0 \leq k \leq n-1 \tag{9}$$

From this formula, it proves that, by the partial evaluation of $peval_{I^k}$ with P_{k+1} given, $peval_{I^{k+1}}$

will be obtained. The formula indicates the systematic way to generate partial compiler (Figure

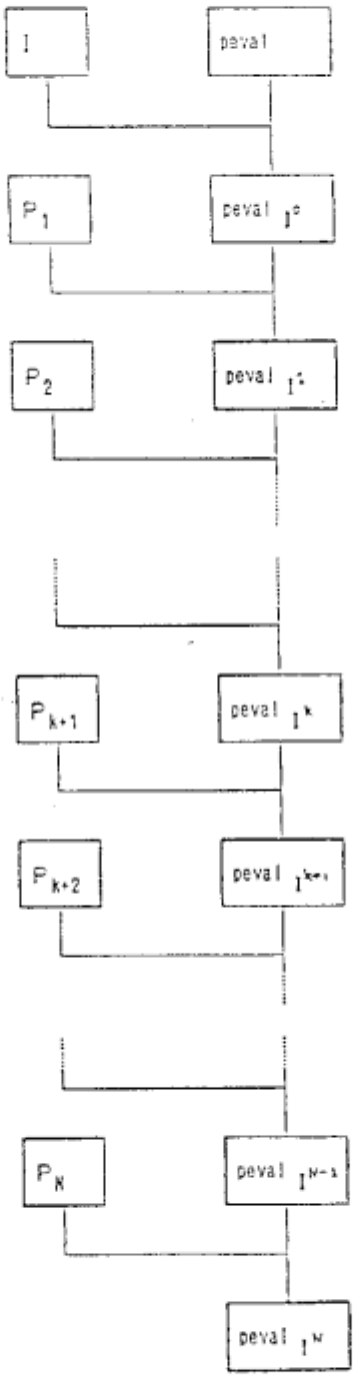

Figure 7. Incremental Specialization of Meta Interpreter

7). In this scheme, we can evolve partial compiler as a new piece of an object program is obtained. The new scheme is more feasible than the previous one which uses compiler-compiler. It is possible to utilize the partial compiler of the arbitrary stage in order to generate the object code of inference system, although it can only infer in the partial program. In this way, the new method for building inference system can be also applicable to the situation where the system acquires inference rules incrementally.

6 CONCLUSION

In this paper, Prolog implementation of the partial evaluation of Prolog program was described and its application to meta programming was investigated. The other partial evaluator of Prolog programs was implemented by Komorowski (Komorowski81) in QLOG. The advantage of Prolog implementation is the ability to partial-evaluate the partial evaluator by itself.

Meta programming becomes to play an important role in Prolog programming because of its expressive power. Partial evaluation will make the meta programming more practical by improving efficiency of meta programs. The importance of meta programming becomes to be recognized also in the parallel logic programming languages such as GHC (Ueda85), PARLOG (Clark84) and Concurrent Prolog (Shapiro83c). Our approach to meta programming using partial evalaution is also promising approach in these languages for achieving both expressive power and efficiency.

Furthermore we have shown the new methodology for building inference systems and demonstrated it by 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 which was constructed incrementally. This feature matches the incremental acquisition of rules in the inference system.

ACKNOWLEDGMENTS

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 Kondou, Masaru Ohki, Hajime Kitakami and other ICOT research staffs who participated in discussions.

REFERENCES

[**Bowen83**] K.Bowen, R.Kowalski: *Amalagamating Language and Metalanguage in Logic Programming*, In K.Clark and S.Tarnlund (eds.) Logic Programming, Academic Press (1983)

[**Bundy81**] A.Bundy, B.Welham: *Using Meta-level Inference for Selective Application of Multiple Rewrite Rules in Algebraic Manipulation*, Artificial Intelligence 16 (1981)

[**Clark82**] K.Clark, F.McCabe: *Prolog: A Language for Implementing Expert Systems*, In D.Michie and Y.H.Pao (eds.) Machine Intelligence 10 (1982).

[**Clark84**] K.Clark, S.Gregory: *PARLOG: Parallel Programming in Logic*, Research Report DOC 84/4, Imperial College, April (1984)

[**Futamura83**] Y.Futamura: *Partial Computation of Programs*, Journal of IECE of Japan, Vol.66, No.2 (1983)

[Hammond83] P.Hammond et al.: *A Prolog Shell for Logic Based Expert Systems*, Imperial College (1983)

[Komorowski81] J.Komorowski: *A Specification of Abstract Prolog Machine and its application to Partial Evaluation*, Linkoping studies in Science and technology dissertations No.69 (1981)

[Matsumoto83] Y.Matsumoto et al.: *BUP: A Bottom-Up Parser Embedded in Prolog*, New Generation Computing, Vol.1, No.2 (1983)

[Shapiro83a] E.Shapiro: *Algorithmic Program Debugging*, The MIT press, 1983

[Shapiro83b] E.Shapiro: *Logic Programs with Uncertainties: A Tools for Implementing Rule-based Systems*, Proc. of IJCAI'83, 1983

[Shapiro83c] E.Shapiro: *A Subset of Concurrent Prolog and Its Interpreter*, ICOT Technical Report TR-003 (1983)

[Takeuchi85] A.Takeuchi: *An Application of Partial Computation to Meta Programming*, Japan Information Processing Society WG memo 85-SF-13, June 1985

[Tanaka84] H.Tanaka: *Rule based Knowledge Representation in Prolog and its Application*, IECE of Japan WG memo AL84-48 1984

[Ueda85] K.Ueda: *Guarded Horn Clauses*, ICOT Technical Report TR-103 (1985)