ICOT Technical Report: TR-456

TR-456

Deriving an Efficient Production System by Partial Evaluation

by
K. Furukawa & H. Fujita

March, 1989

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

Deriving an Efficient Production System by Partial Evaluation

Koichi Furukawa and Hiroshi Fujita

Institute for New Generation Computer Technology,
1-4-28 Mita, Minato-ku, Tokyo, 108, Japan

Toramatsu Shintani

IIAS-FUJITSU LIMITED
140 Miyamoto, Numazu-shi, Shizuoka, 410-03, Japan

ABSTRACT

A great deal of research has been done on applying partial evaluation for optimizing meta-programs in Prolog, such as a rule interpreter with a certainty factor, a bottom-up parser and a formula manipulation system. It has been claimed that the technique is very useful in developing various inference engines for knowledge based systems. However, nobody has succeeded in deriving an efficient production system (PS) through partial evaluation. This paper presents the derivation of an efficient PS by partially evaluating a simple PS interpreter, given a set of rules. The derived codes are shown to be very similar to the compiled codes given by [Shintani 88].

1. Introduction

The combination of rule interpretation and partial evaluation has been noted as a promising approach in developing various inference engines for knowledge based systems. It replaces the rule compiling approach, which has dominated so far. The new approach divides the whole task of compiling into two subtasks: rule interpretation and its partial evaluation. This division makes the entire task much more understandable than the original compilation approach. Furthermore, it enables very easy maintenance, such as modification and debugging.

It should be noted that logic programming supported this approach for two reasons: ease of writing interpreters by meta-programming and ease of developing partial evaluation techniques in logic programming [Komorowski 82],[Venken 84],[Gallagher 86],[Takeuchi 86]. Many applications have been developed using this approach; examples are a simple rule interpreter with a certainty factor, a bottom-up parser and a formula manipulation system. However, the applicability of the technique was rather limited in terms of control structures: if the target interpreter has a control structure which is very different from that of Prolog, it becomes very difficult to optimize it by partial evaluation. The only exception so far was a bottom-up interpreter [Takeuchi 85].

There was an attempt to derive an efficient PS through partial evaluation [Takeuchi 87], but it failed to derive a PS as efficient as that based on the Rete algorithm. This paper presents a new technique for deriving an efficient PS. The technique depends on a PS algorithm developed by [Shintani 88]. In other words, we succeeded in developing a compiler equivalent to Shintani's by the combination of a simple PS interpreter and its partial evaluation.

Section 2 gives a simple PS interpreter in Prolog together with partial evaluation of the recognize predicate. Section 3 describes a working memory driven PS interpreter and its partial evaluation. Section 4 looks at further optimization. Section 5 deals with conflict resolution programs in Prolog. Section 6 gives the performance evaluation results, showing how much the partial evaluation improves the execution time of the PS. Section 7 is the conclusion.

2. Simple Production System Interpreter

Let us first consider a simple production system (PS) without conflict resolution handling. It is well known that the fundamental control structure of a PS is the recognize-act cycle, and it is easy to define a PS interpreter in Prolog using the meta-programming technique shown in Fig. 1.

The `prodSystem` predicate has three arguments: the first argument represents the current working memory (`WM`), the second the final state (`FinalState`) and the last a sequence of applied rules' identifiers for reaching `FinalState`. The definition of `prodSystem` expresses the recognize-act cycle using tail recursion. The first clause of `prodSystem` is for termination.

The `recognize(WM,RuleId,RHS)` predicate means that a production rule whose identifier is `RuleId` is recognized at the current `WM`, and its right hand side is `RHS`. `recognize` first picks up a production rule from the clause database. Then, it tries to prove its left hand side, `LHS`, in the current `WM`, by `deduce(LHS,WM)`. It is assumed that there are only two types of goals on the `LHS`. They are either goals for calling arbitrary Prolog programs, which are in the form of `call(...)`, or literals whose existence is checked in the current working memory.

The `act(RHS,WM,NewWM)` predicate means that the result of applying the `RHS` to the current `WM` is `NewWM`. We assume that the only action command against the working

```
:- op(200,xfx,':').
:- op(150,xfy,=>).

prodSystem(WM,FinalState,[]) :-
        member(FinalState,WM).
prodSystem(WM,FinalState,[RuleID|AppliedRules]) :-
        recognize(WM,RuleID,RHS),
        act(RHS,WM,NewWM),
        prodSystem(NewWM,FinalState,AppliedRules).

recognize(WM,RuleID,RHS) :-
        rule(RuleID : LHS => RHS),
        deduce(LHS,WM).

deduce([],WM).
deduce([C|Cs],WM) :- deduce1(C,WM),
                     deduce(Cs,WM).

deduce1(call(X),_) :- call(X).
deduce1(X,[X|_]) :- member(X,WM).

member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

act([],WM,WM).
act([Act|As],WM,New_WM) :-
        act1(Act,WM,Int_WM),
        act(As,Int_WM,New_WM).

act1(replace(X,Y),[],[]).
act1(replace(X,Y),[X|L],[Y|L]).
act1(replace(X,Y),[Z|L],[Z|L1]) :-
        act1(replace(X,Y),L,L1).
act1(call(X),WM,WM) :- call(X).
```

Fig. 1 A simple PS interpreter without conflict resolution handling.

memory is `replace(X,Y)` which means to replace an element `X` of the current working memory to `Y` (`call(...)` is allowed in `RHS` also).

Although this interpreter is very natural in its behavior, it is hard to make it efficient by partial evaluation. To prove this fact, let us try to partially evaluate the interpreter, given a set of production rules for solving the eight-puzzle. Since production rules are called from the `recognize` predicate, we concentrate on the partial evaluation of `recognize`, defined as:

```
recognize(WM,RuleId,RHS) :-
        rule(RuleId : LHS => RHS),
        deduce(LHS,WM).
```

Assume that we have the following rule:

```
rule(testTile0:
        [goal(putTile0),t(0,0)]
        =>
        [replace(goal(putTile0),goal(putTile1))]).
```

This rule is a rule for testing whether tile 0 is put in the right position or not. If it is, then the working memory element `goal(putTile0)` is replaced by `goal(putTile1)`. Given this rule, the `recognize` clause is specialized as the following clause:

```
recognize(A,testTile0,
        [replace(goal(putTile0),goal(putTile1))]) :-
        deduce([goal(putTile0),t(0,0)],A).
```

As a result, we will have a set of recognize clauses corresponding to each production rule. Since there is no information for selecting the most appropriate `recognize` clause from them in run time, this specialization does not change the essential strategy of testing rules one by one from the beginning until a recognizable rule is found.

3. Working Memory Driven PS Interpreter

To significantly improve performance by partial evaluation, we need to introduce the working memory driven rule selection strategy (we assume that we do not have any negative pattern on LHSs). A new interpreter with this feature is obtained by modifying the `prodSystem` and `recognize` predicates, as shown in Fig. 2.

To add the working memory driven feature, an extra argument is added to the `recognize` predicate. It is in the form of `recognize(Fact,WM,RuleId,RHS)`, which means that a rule named `RuleId` containing a fact, `Fact`, on its `LHS` is recognized in the current `WM`. The new argument, `Fact`, is a member of the current `WM`, as defined in the new `prodSystem` predicate. The new argument is used to filter production rules which do not contain the fact on their `LHS`. The filtering is done by the `del(X,LHS,NewLHS)` predicate, which means that the result of deleting element `X` from list `LHS` is `NewLHS`. The deleted element is unified with a given fact, `Fact`, for filtering. If the unification succeeds, then we need only deduce the rest of the `LHS`, that is, `NewLHS`. Note that this strategy works even if we have negative patterns in condition parts, as long as every rule has at least one positive pattern on its LHS.

```
prodSystem(WM,FinalWM,[RuleId|AppliedRules]) :-
        member(Fact,WM),
        recognize(Fact,WM,RuleId,RHS),
        act(RHS,WM,NewWM),
        prodSystem(NewWM,FinalWM,AppliedRules).
prodSystem(FinalWM,FinalWM,[]).

recognize(Fact,WM,RuleId,RHS) :-
        rule(RuleId:LHS=>RHS),
        del(Fact,LHS,NewLHS),
        deduce(NewLHS,WM).

del(X,[X|Y],Y).
del(X,[A|Y],[A|Z]) :- del(X,Y,Z).
```

Fig. 2 Working memory driven PS interpreter without conflict resolution handling

Interpretive execution cannot benefit from this trick since it still needs to test rules one by one. On the other hand, the filtering can be done at partial evaluation time, and the residual program after partial evaluation does not contain any cases that would be filtered by the lack of any WM element in its LHS during `recognize` predicate execution.

Let us lookat the partial evaluation process of the new PS interpreter, given a set of rules. Like the previous case, the `recognize` predicate

```
recognize(Fact,WM,RuleId,RHS)) :-
        rule(RuleId:LHS=>RHS),
        del(Fact,LHS,NewLHS),
        deduce(NewLHS,WM).
```

is specialized for the given rule set. Since `rule` is defined as a set of facts, the `recognize` clause above is unfolded at `rule` as before.

Then, `del`, defined as:

```
del(X,[X|Y],Y).
del(X,[A|Y],[A|Z]) :- del(X,Y,Z).
```

becomes unfoldable, because `LHS` in the goal `del(Fact,LHS,NewLHS)` is now instantiated to a fixed length list with which `del` may recur.

After that, `deduce(NewLHS,WM)` in turn becomes the candidate for being unfolded. However, it should not be unfolded because `WM` is still unbound in partial evaluation time. Thus `deduce` calls are made residual (more precisely, the goal not to be unfolded is not `deduce`, but `deduce1`).

For instance, given the following `rule`:

```
rule(testTile0:
        [goal(putTile0),t(0,0)]
        =>
        [replace(goal(putTile0),goal(putTile1))]).
```

the `recognize` clause is specialized as two clauses for `testTile0` rule due to the non-determinacy of `del`:

```
recognize(goal(putTile0),A,testTile0,
        [replace(goal(putTile0),goal(putTile1))]) :-
        deduce1(t(0,0),A).

recognize(t(0,0),A,DAtestTile0,
        [replace(goal(putTile0),goal(putTile1))]) :-
        deduce1(goal(putTile0),A).
```

In normal execution, the input/output mode for `recognize` is `(+,+,-)`; hence, `recognize` should become evaluable only when the first and second arguments are instantiated. However, the intended input/output mode for `recognize` is really irrelevant in partial evaluation time. The backward propagation of the instantiation of the first argument caused by calling `rule` and `del` under an unintended input/output mode contributes to the increase in efficiency of the resultant codes despite the decreased space efficiency caused by the increased number of clauses. Note that partial evaluation does not filter irrelevant rules. Instead, the backward propagation causes the same effect as filtering. Although we have excluded irrelevant combinations of the first argument, Fact, and rules to be selected, the vast number of new `recognize` clauses may give rise to a new problem for selecting an appropriate rule.

This problem is solved if the underlying Prolog system supports clause indexing for the recognize clauses, given their first arguments, but in fact, clause indexing is not essential for solving this problem. By introducing a new predicate for each distinct value of the first argument of the `recognize` clauses, the same good performance is obtained. This predicate introduction can be easily done by applying folding.

The question is how far the efficiency will be improved. If we assume that the number of candidate rules containing one or more current working memory elements is roughly constant for any working memory state, then the residual program achieves time complexity independent of the rule set size. Since the original interpretation requires time complexity proportional to the rule set size, n, this means that we can expect the speed of programs to be increased to the degree that the order of time complexity is reduced from $O(n)$ to $O(1)$. This is contrary to the common belief that partial evaluation cannot reduce the order of time complexity.

There remains room for further improvement of the `recognize` predicate by sorting the working memory in terms of recency of updated time. The reason why this modification brings improvement is that changes of the working memory often make some rules newly recognizable. The performance measurement result in Section 6 is based on the improved version.

4. Further Optimization

The recognized rule by `recognize` is represented with its whole RHS which is instantiated according to LHS satisfaction. Hence, heads of specialized `recognize` clauses tend to have a large structure after unfolding `rule` as the result of backward substitution. This may lead an extra consumption of code space and also be a heavy load on head unification of `recognize` at runtime.

However, it is sufficient to have a rule ID and a set of variables which will be instantiated by positively matched elements in the WM in order to represent the recognized rule and to obtain the corresponding instantiation of the RHS for action. Appendix 3 shows a rule description which includes a variable list for this purpose. Using the variable list, heads of specialized `recognize` clauses can be made more compact, thereby saving space and time further.

There is another possibility for optimization. The act part usually contains a lot of computation other than WM operation. Such auxiliary action is allowed through `call(X)` on both the LHS and RHS of a rule. Thus, a program for `X` becomes the target of partial evaluation. The figures shown in Section 6 demonstrate that significant speedup is achieved by this optimization.

5. Conflict Resolution

The simple PS interpreter discussed above takes the first recognized rule as the next rule to be fired and never considers other candidate rules recognizable in the same cycle. It will work for some applications like the eight-puzzle and Rubik's cube problems.

However, the simple PS interpreter can be extended to deal with a *conflict set* and to incorporate *conflict resolution strategies* such as the conventional LEX and MEA strategies used in the OPS5 family.

Conventional strategies for conflict resolution are based on several criteria on matched elements and the LHS of a recognized rule such as:

- Recency (how recently has the element been added to WM?)

- Significance (How significant is the element?)
- Support (how many elements support the LHS?)
- Complexity (how complex is the matched LHS?)

The recency ordering can be implemented easily by employing a list structure for the WM. A new element is placed at the head of the list. The `recognize` predicate picks up elements starting at the head of the list; hence, the recognizable rules are automatically ordered according to the recency of matched elements. That is, a rule which is (positively) supported by a more recent element is recognized earlier.

In many applications, there are some special elements in working memory which are far more significant than other elements. For instance, `goal(_)` in the monkey and banana problem (see Appendix 3) can be taken as the most significant element. A rule which is supported by a more significant element should be recognized earlier. An easy way to implement this strategy is to force the user to supply some declaration such as:

```
more_significant(goal(_),_).
```

According to this ordering declaration, a new element should be inserted in the appropriate position in the WM list.

An implementation of a PS interpreter incorporating the conflict resolution strategy is given in Appendix 1. This interpreter runs the monkey and banana Problem [Brownston 85] in the same way as OPS5 under the MEA strategy.

Note that there may be possibilities of specialization of codes for conflict resolution. For instance, `length` in the `select` predicate in Appendix 1 can be calculated when a rule is given.

6. Performance Evaluation

We measured performance improvement by partial evaluation using the Rubik's cube example. We divided the entire problem into five subproblems: "perfect front edges", "perfect front face", "two perfect layers", "perfect top corners" and "perfect cube". The first row gives the number of rules required to solve each subproblem. Note that any subproblem contains its left subproblem. The second row shows the computation times required to solve each subproblem using the naive production system interpreter without any partial evaluation. The third row shows the computation times after optimizing the call predicates in the action part. The main call predicate is to compute the next cube state by applying a given sequence of operations. Fig. 3 shows the improvement ratio representing the factor of improvement by optimizing call predicates. This graph

Table 1 Performance results of the Rubik's cube problem

Stage	Front edges	Front face	Two layers	Top corners	Finish
Number of rules	15	27	33	40	61
(1) Naive PS	400	600	1119	1520	2659
(2) Optimized action	280	520	879	1219	1879
(3) Specialized recognize	219	299	400	479	560

(1-3) CPU-time (msec) by SICSTUS-Prolog on SUN-3

shows that the performance speeds up by about 1.2 to 1.4 times more than the original program. The last row gives the performance results for the further optimized program by partially evaluating the recognize part. Fig. 4 shows the improvement ratio, representing the factor of improvement given by further optimizing the recognize part. This graph shows that the improvement is linear to the size of the rule number which was predicted in Section 3. In total, we obtained the performance improvement of 1.8 to 4.7 times, depending on the number of rules.

7. Comparison with Related Research

This section compares our method with the RETE and TREAT algorithms. As stated earlier, our new method adopts Shintani's algorithm. The difference between them is the realization method of the compiler. Both the RETE and TREAT algorithms store a set of working memory elements to a memory (called the alpha memory) associated with each of the left hand side elements (condition elements) of each rule. This set is an answer set for the database query consisting of a condition element regarding the working memory as a relational database [Miranker 84]. Our algorithm, however, does not maintain any partial matching results. Instead, it associates a set of candidate rules with each possible pattern of working memory elements. By this association, we can quickly access relevant rules which may become recognizable after the change of a working memory. This corresponds to the feature of TREAT which calculates the derivative of the conflict set from a seed. The main difference between our algorithm and TREAT is that we do not perform any join to calculate the derivative. Instead, we perform test operations for each remaining condition of the candidate rules. A set of all rule instantiations recognizable for the current working memory is obtained by combining compulsory fail and the backtracking mechanism in Prolog. A more detailed performance comparison is left for future research.

8. Conclusion

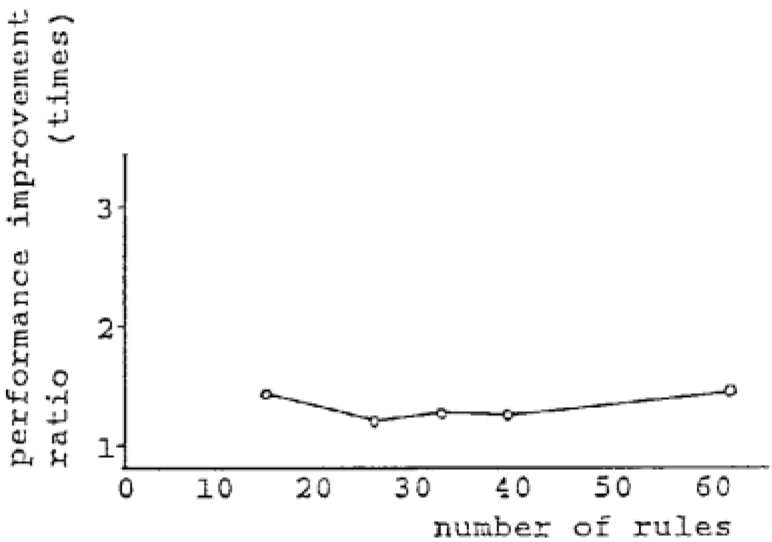

Fig. 3 Performance improvement brought by the act part optimization

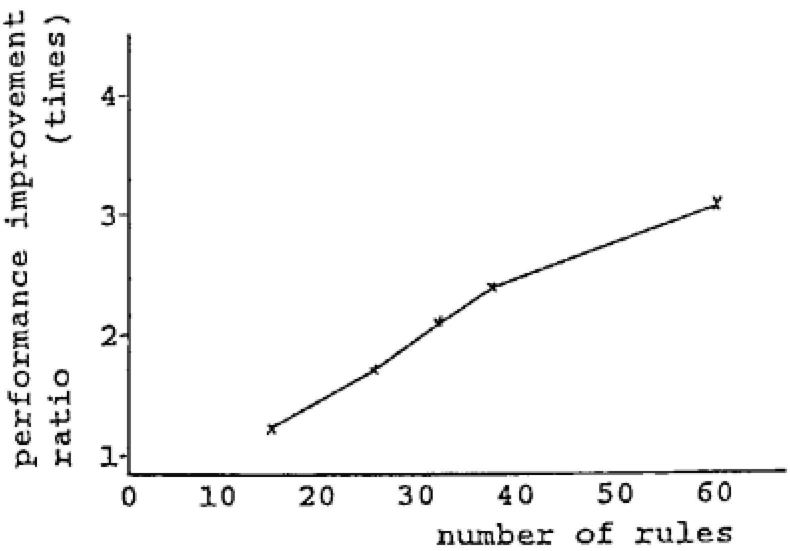

Fig. 4 Performance improvement brought by the recognize part specialization

This paper presented an approach to developing a production system compiler by combining a PS interpreter and a general partial evaluator. We developed a working memory driven interpreter to obtain maximum performance improvement by partial evaluation. The residual codes after partial evaluation were shown to be equivalent to those obtained by a compiler developed by [Shintani 88].

The performance improvement is about two times when the rule set is small and without conflict resolution. We also showed that we can expect the speed of programs to be increased to the degree that the order of time complexity is reduced from $O(n)$ to $O(1)$. This is contrary to the common belief that partial evaluation cannot reduce the order of time complexity.

We also developed a complete version of the PS interpreter which handles conflict resolution, negative patterns, and delete commands for working memory update. Performance improvement of the complete PS was about 30 % speedup (less than one second of CPU time) for a sample problem, monkey and banana [Brownston 85].

Our production system program does not use any assert/retract primitives for representing the working memory and conflict set. Furthermore, the program structure is simple enough to determine input/output modes for each predicate argument, and therefore it seems possible to transform it into an equivalent FGHC program by applying Ueda's transformation method [Ueda 86]. This further transformation is a future research subject.

Acknowledgement

References

[Brownston 85] Brownston, L., Farrell, E. K., and Martin, N., "Programming expert system in OPS5", Addison-Wesley, 1985.

[Bekke 86] Brekke, B., "Benchmarking Expert System Tool Performance", Ford Aerospace Tech. Note, 1986.

[Forgy 81] Forgy, C. L., "OPS5 User's Manual", CMU-CS-81-135, July, 1981.

[Forgy 82] Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence 19, pp.17-37, 1982.

[Fujita 88] Fujita, H. and Furukawa, K., "A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation", Proc. of Workshop on Partial Evaluation and Mixed Computation, Gl. Avernes, Denmark 1987, New Generation Computing, Vol. 6, Nos. 2, 3, 1988.

[Futamura 83] Futamura, Y., "Partial Computation of Programs", Lecture Notes in Computer Science 147, Springer Verlag, 1983.

[Gallagher 86] Gallagher, J., "Transforming Logic Programs by Specialising Interpreters," In *ECAI-86 7th European Conference on Artificial Intelligence, Brighton Centre, United Kingdom*, pp.109-122, 1986.

[Komorowski 82] Komorowski, H.J., "Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog," In *Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pp.255-267, 1982.

[Kowalski 79] Kowalski, R., "Logic for Problem Solving", Elsevier North Holland, 1979.

[Lesser 77] Lesser, V. R. and Erman, L. D., "A Retrospective View of the HEARSAY-II Architecture", Proc. Fifth IJCAI, pp.790-800, 1977.

[Matsumoto 83] Matsumoto, Y., Tanaka, H., and Kiyono, M., "BUP: A Bottom-up Parser Embedded in Prolog", New Generation Computing 1, No. 2, 1983.

[McDermott 78] McDermott, J., Newell, A., and Moore, J., "The Efficiency of Certain Production Implementations", in Pattern Directed Inference Systems (Waterman, D. A. and Hayes-Roth, F., eds.), Academic Press, pp.155-176, 1978.

[Miranker 87] Miranker, D. P., "TREAT: A Better Match Algorithm for AI Production Systems", AAAI-87, pp.42-47, 1987.

[Mizoguchi 83] Mizoguchi, F., Miwa, K., and Honma, Y., "An Approach to PROLOG Basis Expert System", Proc. of the Logic Programming Conference '83, Tokyo, March, pp.22-24, ICOT, 1983.

[Nayak 88] Nayak, P., Guputa, A., and Rosenbloom, P., "Comparison of Rete and Treat Production Matchers for Soar (A Summary)", AAAI-88, pp.693-698, 1988.

[Sgintani 86] Shintani, T., Katayama, Y., Hiraishi, K., and Toda, M., "KORE: A Hybrid Knowledge Programming Environment for Decision Support based on A Logic Programming Language", Lecture Notes in Computer Science 264, pp.22-33, 1986.

[Shintani 88] Shintani, T., "A Fast Prolog based Production System KORE/IE", Proc. of the Fifth International Conference and Symposium on Logic Programming (Kowalski, R. A. and Bowen, K. A., eds.), MIT Press, pp.26-41, 1988.

[Takeuchi 86] Takeuchi, A and Furukawa, K., "Partial Evaluation of Prolog Programs and Its Application to Meta Programming", Information Processing 86 (Kuger H. J. ed.), Dublin, Ireland, pp.415-420, North-Holland, 1986.

[Takeuchi 87] Takeuchi, A and Fujita, H., "Competitive Partial Evaluation - Some Remaining Problems of Partial Evaluation", Proc. of Workshop on Partial Evaluation and Mixed Computation, Gl. Avernes, Denmark 1987, New Generation Computing, Vol. 6, Nos. 2, 3, 1988.

[Venken 84] Venken, R., "A Prolog Meta-interpreter for Partial Evaluation and Its Application to Source to Source Transformation and Query-Optimization." In O'Shea,T. (ed.), *ECAI-84, Advances in Artificial Intelligence, Pisa, Italy* North-Holland, pp.91-100., 1984.

Appendix 1 PS interpreter with conflict resolution

```
:- op(150,xfy,=>).
:- op(149,xfy,=:).

%% top level loop
prodSystem(WM,FinalState) :-
        member(FinalState,WM).
prodSystem(WM,FinalState) :-
        member(Fact,WM),
        bagof(Rule,recognize(Fact,WM,Rule),CS),
           %% CS: conflict set
        select(CS,Rule1*LHS1),
        act(Rule1,LHS1,WM,NewWM),
        prodSystem(NewWM,FinalState).

%% recognition phase
recognize(Fact,WM,Rule*LHS) :-
        rule(Rule,LHS=>_),
        del(Fact,LHS,RestLHS),
        deduce(RestLHS,WM).

del(X,[X=:V|Y],Y) :- X=V.     %% memorize the matched pattern
del(X,[X|Y],Y).
del(X,[A|Y],[A|Z]) :- del(X,Y,Z).

deduce([C|Cs],WM) :- deduce1(C,WM),
                     deduce(Cs,WM).
deduce([],_).

deduce1(call(X),_) :- call(X).
deduce1(X=:V,WM) :- member(X,WM), X=V.
                              %% memorize the matched pattern
deduce1(X,WM) :- member(X,WM).
deduce1(-X,WM) :- deduce_negative(X,WM).  %% negative pattern

deduce_negative(X,[X|_]) :- !, fail.
deduce_negative(X,[_|WM]) :- deduce_negative(X,WM).
deduce_negative(_,[]).

%% act phase
act(Rule,LHS,WM,NewWM) :-
        rule(Rule,LHS=>RHS),  %% retrieve the corresponding RHS
        act1(RHS,WM,[],NewWM).

act1([call(X)|RHS],WM,AddWM,NewWM) :-
        call(X),
        act1(RHS,WM,AddWM,NewWM).
act1([replace(X,Y)|RHS],WM,AddWM,NewWM) :-
        delete(X,WM,WM1),     %% remove X from WM
        act1(RHS,WM1,[Y|AddWM],NewWM).
                              %% add Y tentatively to AddWM
```

```
act1([+X|RHS],WM,AddWM,NewWM) :-
        act1(RHS,WM,[X|AddWM],NewWM).
                                 %%  add X tentatively to AddWM
act1([-X|RHS],WM,AddWM,NewWM) :-
        delete(X,WM,WM1),        %%  remove X from WM
        act1(RHS,WM1,AddWM,NewWM).
act1([stop|_],_,_,[]).           %%  terminator
act1([],WM,AddWM,NewWM) :-
        add(AddWM,WM,NewWM).     %%  add elements in AddWM to WM

delete(X,[X|WM],WM) :- !.
delete(X,[Y|WM],[Y|WM_]) :- delete(X,WM,WM_).
delete(_,[],[]).

add([X|AddWM],WM,NewWM) :-
        add1(X,WM,WM1),
        add(AddWM,WM1,NewWM).
add([],NewWM,NewWM).

%%  more significant element is to be placed at the head of WM
add1(X,[Y|WM],[X,Y|WM]) :- more_significant(X,Y).
add1(X,[Y|WM],[Y|NewWM]) :-
        more_significant(Y,X), add1(X,WM,NewWM).
add1(X,WM,[X|WM]).

member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

%%  select phase (more complex rule is to be selected)
select([R1*LHS1,R2*LHS2|Rules],Rule) :-
        length(LHS1,L1), length(LHS2,L2), L1=<L2,
        select([R2*LHS2|Rules],Rule).
select([R1,_|Rules],Rule) :-
        select([R1|Rules],Rule).
select([Rule],Rule).
```

Appendix 2   A rule from the monkey and banana problem

```
rule('Holds::Object-Ceil',
      [ goal(active,holds,O1,On_g,To) =: Goal,
        phys_object(O1,P,light,ceiling) =: Object,
        phys_object(ladder,P,_,floor),
        monkey(At_m,ladder,nil) =: Monkey,
        - phys_object(_,_,_,O1) ]
   =>
      [ call((nl,write('Grab '),write(O1),nl)),
        replace(Goal,goal(satisfied,holds,O1,On_g,To)),
        replace(Monkey,monkey(At_m,ladder,O1)),
        replace(Object,phys_object(O1,P,light,nil)) ]).
```

Appendix 3 A rule with a variable list

```
rule('Holds::Object-Ceil'(A,B,C,D,E,F,G,H,I,J),
      [ goal(active,holds,A,B,C) =: D,
        phys_object(A,E,light,ceiling) =: F,
        phys_object(ladder,E,G,floor),
        monkey(H,ladder,nil) =: I,
        - phys_object(J,K,L,A) ]
   =>
      [ call((nl,write('Grab '),write(A),nl)),
        replace(D,goal(satisfied,holds,A,B,C)),
        replace(I,monkey(H,ladder,A)),
        replace(F,phys_object(A,E,light,nil)) ]).
```