TM-0484

Abstract Interpretation and Partial Evaluation of Prolog Programs

by

H. Fujita

March. 1988

Abstract Interpretation and Partial Evaluation of Prolog Programs

Hiroshi FUJITA

First Research Laboratory, ICOT

Sep 9, 1986
Revised Oct 17, 1986
Reprinted March 11, 1988

ABSTRACT

This article deals with abstract interpretation and its application to some preanalyses for partial evaluation of Prolog programs. Our experiences with the previous version of partial evaluator showed us just how much a human insight is needed to develop a partial evaluator that terminates with successful result. In order to make partial evaluation really valuable, we have to automate it. For that purpose, we constructed yet another partial evaluator which uses information given by the preanalysis besed on abstract interpretation. Although the method described in this article is not sufficient enough to fully automate the partial evaluator, the more sophisticated preanalysis we make on the basis of abstract interpretation, the more powerful partial evaluator we can develop.

1 Introduction

Our experiences with Takeuchi's partial evaluator [Takeuchi 86] for (pure) Prolog programs (abbreviated to TPE below) showed us just how much a human insight is needed to develop a partial evaluator that terminates with a successful result.

The directives to TPE are given either in advance by auxiliary clauses or interactively by the user during partial evaluation (possibly in both ways). At any rate, we felt that the burden on the user was too heavy involving difficult problems even in small applications. We have to minimize this burden as far as possible in partial evaluation to make it really valuable. The attempt to find ways to automate partial evaluation of Prolog programs is a responce to this.

We constructed yet another partial evaluator, rather different from TPE, that is simple and powerful enough to output the same (possibly better) results as TPE. We begin by describing the new partial evaluator in the next section, introducing some criteria for *evaluability/unfoldability*, which directs processing by our partial evaluator. In section 3, we introduce abstract interpretation, which will be applied to our partial evaluator as a method for preanalyses in Section 4, and finally, we make some concluding remarks.

2 Partial Evaluation

Partial evaluation in the case of Prolog is concerned with such computations as unification (constant propagation), subgoal expansion (unfolding) and evaluation of evaluable system predicates, all of which are normally executed by the Prolog interpreter at runtime.

Of course, we can partially evaluate any Prolog program at any time by unfolding it at any goal in a program clause body at any depth. However, reckless unfolding (or uncontrolled partial evaluation) does not have beneficial results. In general, partial evaluation is useful only when we have a goal that has enough information ready for execution. This also gives us another seed for partial evaluation at other places propagating information via instantiation of shared variables, narrowing search spaces or providing shortcut paths which would be traversed at runtime.

We can think of several situations in which partial evaluation may become effective, such as

(1) An application program supplied with (incomplete) data, or

(2) A meta interpreter supplied with its program, or

(3) A program which is partially evaluable itself due to somewhat redundant description in its source codes is given.

Typical starting points of partial evaluation are a call of a predicate that is defined by unit clauses in case (1), a system call "clause(Head,Body)" and the like in case (2), and a goal of which arguments are instantiated literally to the extent of being evaluable in case (3).

2.1 An Outline of our Partial Evaluator

Our partial evaluator operates under the following two rules:

[**Evaluation Rule**] If a goal appearing in a body of a clause in the program is an instance of a canonical goal marked evaluable, then evaluate it as the usual interpreter does and rewrite the clause to the one that reflects the evaluation.

[**Unfolding Rule**] If a goal that appears in a body of a clause in the program is an instance of a canonical goal marked unfoldable, then unfold it by clauses in the program which defines the goal, and rewrite the clause to forms corresponding to matching clauses by head unification with the goal (as the usual interpreter does).

A canonical goal is a representative goal among variants (identical up to variable renaming), whose variables are thought always fresh (distinct from all other variables mentioned elsewhere in the context).

If a rule is successfully applied to program P_0, the program is rewritten into a new program P_1. A rewrite as the result of one of the rules above would cause successive rewrites. Thus, rewriting is repeated until no rule is applicable to the current program P_n. Note that one of the differences between TPE and our partial evaluator is that the former is driven top-down, while the latter is bottom-up and iterative in nature.

We want the rewritings to terminate and the program to converge. In order to assure the termination, we have to set appropriately *"evaluability/unfoldablity"* conditions. We will investigate evaluability/unfoldability conditions in detail.

2.2 Evaluability/Unfoldability

Evaluability and unfoldability conditions have to be we should carefully arranged so that they guarantee that the partial evaluator never diverges.

Some critera for evaluability and unfoldability are given in the form of these rather ad hoc rules.

"unit" unfoldability A goal unifiable only with (heads of) unit clauses is unfoldable.
This rule may be effective especially in data base applications, although it is likely to run into the danger of spatial explosion by multiplication of unfoldings.

"non-recursive" unfoldability A non-recursive goal, that is, a goal that never calls a subgoal of the same predicate with the same arity, is unfoldable.
This rule includes the previous one as a special case. An unfolding by this rule correspond to inline expansion of a subroutine call by its body in conventional procedural language programs.

"halfway-decomposed" unfoldability A goal that is an instance of a head goal but not of body goals is unfoldable; where a head goal is a canonical goal of which a variant appears at the head of a program clause, and a body goal appears in the body.
Intuitively, the reason why such a goal becomes unfoldable is that it has an argument which has been halfway decomposed, so that we can further decompose it by another step of unfolding.

The above rules are not complete; that is, some unfoldable goals are covered by them, but others are not. We are interested in the last of the unfoldabilities above in particular, because it is concerned with the most typical situation that happens in the partial evaluation of programs which include ordinary recursive definitions of predicates.

Before we elaborate on this type of unfoldability, we introduce abstract interpretation in the next section, which will be used as a general conceptual tool for extracting useful information from programs.

3 Abstract Interpretation

[Cousot 81] states that,

" A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some information on the actual computations. "

After defining several preliminary issues in abstract interpretation, we will elaborate on its application to partial evaluation.

3.1 Outline of Abstract Interpretation

Suppose we have a program and want to focus on some specific aspect of the behavior of the program, which does not necessarily require exact interpretation or execution.

First, we define the domain for each construct of the program according to our concerns. Such a domain may be different from that for the original programs. It requires a lattice structure, that is, the values in the domain must be partially ordered with bottom (infimum) and/or top (supremum) elements. Next, we extract the equations in the domain from each line of the program text which reflect the behavior of the program only through the values in the domain for each construct of the program.

We can now perform abstract interpretation quite algorithmically by iterative method as follows.

(1) Assign the initial value to each construct of the program which appear at the right hand sides of the equations.

(2) Compute the new value for each construct of the program which appears at left hand side of an equation using the old values already assigned to each construct of the program at right hand side of the equation.

(3) Repeat step (2) until all the values for the program constructs with which the set of equations is concerned are converged, that is, when no new value update is possible.

Thus, once we set an appropriate system of equations, extracted from program constructs, we can mechanically get the information we wanted by solving the system of equations with respect to the unknowns by iterative methods.

3.2 Abstract Interpretation in Prolog

In the area of logic programming, the work of [Mellish 86] can be cited which is mainly concerned with the following three applications.

- Mode declarations (information on instantiation of the number of solutions that predicates can produce)

- Determinacy information (information about the number of solutions that predicates can produce)
- Information about shared structures (this can be used, for instance, to indicate places where "occur check" might be desirable)

Our approach is almost the same as [Mellish 86], in that we apply abstract interpretation in a manner which may deviate somewhat from formal treatments.

4 Derivation of Unfoldability

Some (not all) of the conditions for partial evaluation may be automatically derived by preanalizing given program clauses. Such preanalyses are done by several variations of abstract interpretation [Mellish 86], [Mellish 85]. As an illustration of abstract interpretation, we begin with a simple cross reference analysis.

4.1 Cross Reference Analysis

P denotes the set of predicates of goals that will possibilly be called in the course of Prolog execution under the goal of predicate P. For each predicate H defined by several clauses $\{H :- B_1, \ldots, B_n\}$ in a given program, the following equation is derived

$$\begin{aligned}\mathbf{H} \ = \ & \mathbf{B}_{1,1} \cup \mathbf{B}_{1,2} \cup \ldots \cup \mathbf{B}_{1,n_1} \cup \\ & \ldots \\ & \mathbf{B}_{m,1} \cup \mathbf{B}_{m,2} \cup \ldots \cup \mathbf{B}_{m,n_m}\end{aligned}$$

$\mathbf{P} \cup \mathbf{Q}$ denotes the operation that produces the join of the sets of values $\{P_i\}$ and $\{Q_i\}$, the values for **P** and for **Q**. The values for **P** with the usual subset ordering are structured by the lattice shown in Figure 1.

We solve the system of equations by iteratively updating the value for **P**; that is, starting from the initial value assigned to each **P**, we evaluate the righthand side of each equation with the old values to generate a new value for **P** at the lefthand side, and repeat the process until all values for **P** converge.

Initial value for each P is set ϕ: the infimum of the lattice. The supremum of the lattice is the set of all predicates appearing in given program clauses. Since the lattice is finite and the operation $\cup$ is order-preserving, the iteration will eventually terminate in a

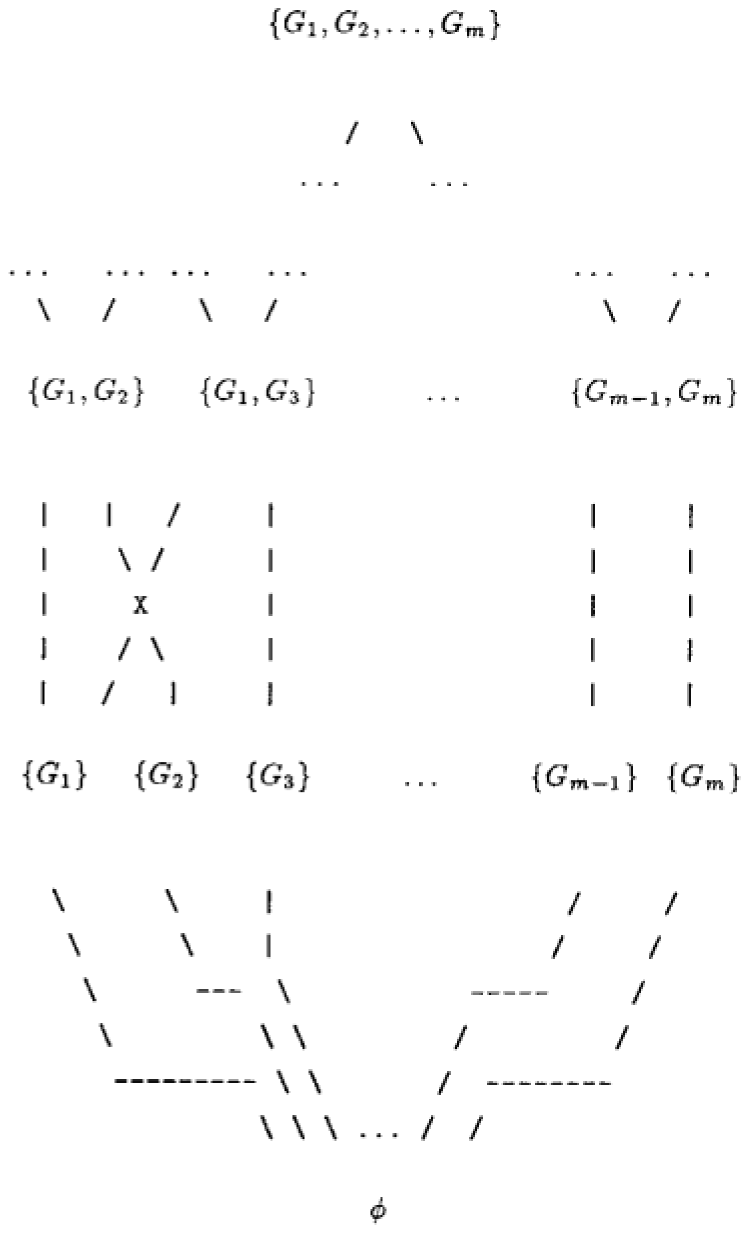

Figure 1: Thc Lattice for Cross Reference Analysis

convergence (fixpoint).

To illustrate the analysis, suppose that we have the following program:

$$ancestor(X,Y) :-\ parent(X,Y).$$
$$ancestor(X,Y) :-\ parent(X,Z),\ ancestor(Z,Y).$$
$$parent(X,Y) :-\ father(X,Y).$$
$$parent(X,Y) :-\ mother(X,Y).$$

We derive the following equations:

$$\begin{aligned}\mathbf{ancestor/_2} &= \mathbf{parent/_2} \cup \mathbf{ancestor/_2}\\ \mathbf{parent/_2} &= \mathbf{father/_2} \cup \mathbf{mother/_2}.\end{aligned}$$

The initial values are

$$\mathbf{ancestor/_2} = \mathbf{parent/_2} = \mathbf{father/_2} = \mathbf{mother/_2} = \phi.$$

After the first iteration we get

$$\begin{aligned}\mathbf{ancestor/_2} &= \{parent/_2, ancestor/_2\}\\ \mathbf{parent/_2} &= \{father/_2, mother/_2\}.\end{aligned}$$

After the second iteration we get

$$\begin{aligned}\mathbf{ancestor/_2} &= \{parent/_2, ancestor/_2, father/_2, mother/_2\}\\ \mathbf{parent/_2} &= \{father/_2, mother/_2\}.\end{aligned}$$

The third iteration does not change the old values, so the iteration has converged. The final value of $\mathbf{ancestor/_2}$ says that a goal of predicate $ancestor/_2$ may invoke as subgoals $parent/_2$, $ancestor/_2$, $father/_2$ and $mother/_2$ in execution.

The inverse relation, that is the "called-by" relation instead of the "call" relation, is also derived in the same way. Note that the first iteration gives just a part of the usual cross reference information, and subsequent iterations compute in fact the transitive closure of this simplest cross reference relation.

A slightly extended version of the above analysis is described as one of our preanalyses for partial evaluation of Prolog programs.

4.2 Extended Cross Reference

We denote a canonical goal with respect to some specific goal G by $\overline{G}$, renaming each variable in G by a number preceded by "@". The equation derived from each clause $H :- B_1, \ldots, B_n$ in given program clauses is

$$\overline{\mathbf{H}} = \overline{\mathbf{B}}_1 \uplus \overline{\mathbf{B}}_2 \uplus \ldots \uplus \overline{\mathbf{B}}_n.$$

The operation $\overline{\mathbf{P}} \uplus \overline{\mathbf{Q}}$ produces the join of the sets of values of any $\overline{\mathbf{R}}$ unifiable with $\overline{\mathbf{P}}$ or $\overline{\mathbf{Q}}$. The lattice is like that in Figure 1, except that more varieties of the form of G than those of the previous example are included.

For instance, suppose we have the following program [Takeuchi 86]:

$$\begin{array}{l}
solve(true, [100]). \\
solve((A, B), Z) :- \; solve(A, X), \; solve(B, Y), \; append(X, Y, Z). \\
solve(not(A), [CF]) :- \; solve(A, [C]), \; C < 20, \; CF \; is \; 100 - C. \\
solve(A, [CF]) :- \; rule(A, B, F), \; solve(B, S), \; cf(F, S, CF)..
\end{array}$$

The derived equation is

$$\begin{aligned}
\overline{\mathbf{solve(true, [100])}} &= \phi \\
\overline{\mathbf{solve((A, B), Z)}} &= \overline{\mathbf{solve(A, X)}} \uplus \overline{\mathbf{solve(B, Y)}} \uplus \overline{\mathbf{append(X, Y, Z)}} \\
\overline{\mathbf{solve(not(A), [CF])}} &= \overline{\mathbf{solve(A, [C])}} \uplus \overline{\mathbf{C < 20}} \uplus \overline{\mathbf{CF\ is\ 100 - C}} \\
\overline{\mathbf{solve(A, [CF])}} &= \overline{\mathbf{rule(A, B, F)}} \uplus \overline{\mathbf{solve(B, S)}} \uplus \overline{\mathbf{cf(F, S, CF)}}.
\end{aligned}$$

The converged values are

$$\begin{aligned}
\mathbf{solve(true, [100])} &= \phi \\
\mathbf{solve((@1, @2), @3)} &= \mathbf{solve(not(@1), [@2])} \;=\; \mathbf{solve(@1, [@2])} \\
&= \{ \; solve(@1, @2), \; solve(@1, [@2]), \\
&\qquad append(@1, @2, @3), \; rule(@1, @2, @3), \\
&\qquad cf(@1, @2, @3), \; @1 < 20, \; @1 \; is \; 100 - @2 \; \}
\end{aligned}$$

4.3 Deriving "halfway-decomposed" Unfoldability

We can derive "halfway-dcomposed" unfoldability for recursive predicates utilizing the above result.

First, we extract head goals and body goals from the extended cross reference. A *head goal* is a canonical goal for the goal that appears at the head position of a clause, while a *body goal* is a canonical goal for the goal that appears at the body position of a clause, after the n-th ($n \geq 0$) unfolding of the clause, given by the extended cross reference analysis as outlined above.

The head goals for $solve/_2$ are

$$\{solve(true,[100]),\ solve((@1,@2),@3),\ solve(not(@1),[@2]), solve(@1,[@2])\}.$$

The body goals for $solve/_2$ are

$$\{solve(@1,@2),\ solve(@1,[@2])\}.$$

Next, we take the difference of the two sets, which is

$$\{solve(true,[100]),\ solve((@1,@2),@3),\ solve(not(@1),[@2])\}.$$

Now, each element in the set above is marked unfoldable following the "halfway-decomposed" unfoldability criterion. That is, head goal that is not also a body goal at the same time can be reduced by several steps of goal reduction ultimately into some of the body goals. Accordingly, it becomes desirable to evaluate a goal which is an instance of a head goal, but evaluation of a body goal is not since it results in the partial evaluation process not terminating..

5 Concluding Remarks

The "evaluability/unfoldability" analysis described above seemed to work fairly well as far as we know from limited examples such as the one listed in appendix.

However, it is not sufficient yet of course. We might be able to cover the evaluabilities/unfoldabilities which could not be caught here by applying other variations of abstract interpretation, such as mode analysis, functionality analysis, determinacy analysis, etc. In particular, the computation of *measured subsets* [Boyer 79] or the like that would be most useful for our application. We have to study all these analyses in detail, then integrate them into a unified analyzer for automating partial evaluation of Prolog programs.

References

[Boyer 79] Boyer, R. S. and Moore, J. S. *A Computational Logic,* Academic Press, 1979.

[Cousot 81] Cousot, P. and Cousot, R.: *"Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Application of Fixpoints"*, Principles of Programming Languages, 1977.

[Mellish 85] Mellish, C. S.: *"Some Global Optimizations for a Prolog Compiler"*, Journal of Logic Programming, Vol. 2, No. 1, 1985.

[Mellish 86] Mellish, C. S.: *"Abstract Interpretation of Prolog Programs"*, International Conference on Logic Programming, 1986.

[Takeuchi 86] Takeuchi, A. and Furukawa, K.: *"Partial Evaluation of Prolog Programs and its Application to Meta Programming"*, Information Processing, 1986.

Appendix A: A Skeleton of Our Partial Evaluator

```
% while an unfixed clause exists at stage I, do peval2
peval :- cl(_,unfixed,I,_,_), peval2(I), abolish_cl(I),!,
   peval.
% where a clause is asserted as
%       'cl(Identifier,Label,Stage,Head,Body)'

% for all clauses at stage I, do peval3
peval2(I) :- I1 is I+1, cl(ID,Label,I,H,B),
   peval3(ID,Label,I,I1,H,B),
   fail ; true.

% case analysis by Label
peval3(ID,fixed(S),I,I1,H,B) :-
   assert(cl(ID,fixed(S),I1,H,B)),!.
peval3(ID,unfixed,I,I1,H,B) :-
   select_goal(B,G), peval4(I,I1,ID,H,G),!.
peval3(ID,_,I,I1,H,B) :-
   assert(cl(ID,fixed(I1),I1,H,B)),!.

select_goal((P,Q),goal(Goal,X,Y)) :-
   select_goal(P,goal(Goal,X,PP)), Y=(PP,Q) ;
   select_goal(Q,goal(Goal,X,QQ)), Y=(P,QQ).
select_goal(B,goal(B,X,X)).
```

```
% evaluation
peval4(_,I1,ID,Head,goal(Goal,NewGoal,NewBody)) :-
   evaluable(Goal),
   ( solve(Goal), NewGoal=true,
     reform_body(NewBody,NB),
     assert(cl(ID,unfixed,I1,Head,NB)) ;
     true ).

solve((P,Q)) :- solve(P), solve(Q).
solve(H) :- cl(_,_,_,H,B), solve(B).
solve(G) :- G.

% unfolding
peval4(I,I1,ID,Head,goal(Goal,NewGoal,NewBody)) :-
   unfoldable(Goal),
   unfold(I,I1,Head,Goal,NewGoal,NewBody).

unfold(I,I1,Head,Goal,NewGoal,NewBody) :-
   cl(_,_,I,Goal,Newgoal),
   reform_body(NewBody,NB),
   new_clause(I1,Head,NB),
   fail ; true.
```

Appendix B: An Example

B.1 Source Program

```
%  Rule oriented inference engine
%
%        Compute Certainty factor
%                                     by A.Takeuchi  [Mar.27-1984]

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]) :- \+A=true, 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).

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

rule(A,B,F) :- ((A:-B)<>F).
rule(A,true,F) :- (A<>F), \+(A=(_:-_)).
```

B.2 Object Rules

```
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.3 Unfoldability Derived by The Preanalysis

```
unfoldable((A<>B)).
unfoldable(append([A|B],C,[A|D])).
unfoldable(cf(A,B,C)).
unfoldable(product([A|B],C,D)).
unfoldable(rule(A,B,C)).
unfoldable(solve(true,[100])).
unfoldable(solve((A,B),C)).
unfoldable(solve(not(A),[B])).
```

B.4 Evaluability Supplied by The User

```
evaluable(solve(true,_)).
evaluable(cf(X,Y,_)) :- ground(X), ground(Y).
evaluable(product(X,Y,_)) :- list(X), ground(Y).
```

B.5 The Result of Partial Evaluation

```
solve(true,[100]).
solve((A,B),C) :- solve(A,D), solve(B,E), append(D,E,C).
solve(not(A),[B]) :- solve(A,[C]), C<20, B is 100-C.

solve(should_take(A,B),[C]) :-
   solve(complains_of(A,D),E),
   solve(suppresses(B,D),F),
   solve(not(unsuitable(B,A)),G),
   append(F,G,H), append(E,H,I), product(I,100,J), C is 70*J/100.

solve(unsuitable(A,B),[C]) :-
   solve(aggravates(A,D),E),
   solve(suffers_from(B,D),F),
   append(E,F,G), product(G,100,H), C is 80*H/100.

solve(suppresses(aspirin,pain),[60]).
solve(suppresses(lomotil,diarrhoea),[65]).
solve(aggravates(aspirin,peptic_ulcer),[70]).
solve(aggravates(lomotil,impaired_liver_function),[70]).
```