TCPW/CW/K.INT.

BIM-PROLOG

Joint Project between
BIM
and
Department of Computer Science
Katholieke Universiteit LEUVEN

Sponsored by DPWB/SPPS
under grant nr KBAR/SOFT/1

A Prolog Meta-interpreter for Partial Evaluation and its Application to Source to Source Transformation and Query Optimisation

by
Raf VENKEN *

Internal Report
BIM-prolog IR3

May 1984

* BIM
Kwikstraat 4
B-3Ø78 Everberg Belgium
tel. +32 2 759 59 25

** Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 2ØØA
B-3Ø3Ø Heverlee Belgium
tel. +32 16 2Ø Ø6 56

DPWB = Diensten van de eerste minister : Programmatie van het Wetenschapsbeleid.
SPPS = Services du premier ministre : Programmation de la Politique Scientifique.

A PROLOG META-INTERPRETER FOR PARTIAL EVALUATION AND ITS APPLICATION TO SOURCE TO SOURCE TRANSFORMATION AND QUERY OPTIMISATION

In this report we investigate how to apply the concepts of partial evaluation on source to source transformations of Prolog program. Starting from a Prolog metainterpreter we gradually construct a partial evaluation system and illustrate its behaviour by several examples. The final system is argued to be applicable to full Prolog programs. In order to achieve this we introduce a special control structure and describe a meta-interpreter that handles Prolog extended with this control structure. Finally we describe the impact a partial evaluation system can have on the optimisation of queries in deductive databases.

1. INTRODUCTION

In [8] partial evaluation of Prolog programs is investigated as a part of a theory of interactive, incremental programming, with the goal to provide formally correct, interactive programming tools for program transformation. Those program transformation tools will play an eminent role in program optimisation, relieving the programmer from concern for efficiency in many cases.

In this report we show how a partial evaluation system for Prolog programs can be build gradually, starting from a Prolog meta-interpreter. The system is argued to work correctly for 'full-Prolog' programs and could be used as an automatic optimisation tool for Prolog program.

Currently we investigate how partial evaluation could be used as an alternative for the different techniques of compiling queries in deductive databases, where the deductive database is build around a Prolog system. The classical compiling techniques restrict to non-recursive databases, where the recent algorithms compile towards iterative programs (see [2], [4] and [6] for more details). We indicate how partial evaluation can be used in this context.

2. OBJECTIVES.

The goal of partial evaluation is to transform programs into more efficient ones. This optimisation is accomplished by mainly three techniques : instantiate the parameters of a program by propagating values for top-level formal arguments through the program (execute the unification at "compile-time"), reduce the number of logical inferences by opening calls and by evaluating builtin predicates (builtins for short) whenever possible.

Let us consider a simple example : a straightforward way to implement a screen management system is to define an abstract data type 'screen' which consists of some fixed text-fields and some input/output fields, and some associated manipulations (here a routine to display the screen on a terminal).

```
          screen(test, 5, 15). {screen test, first line is 5th line
                                 of terminal, last line 15th}

          text(test, 5, 15, 'Screen A program X').
          text(test, 9,  5, 'Command').
          text(test,11,  5, 'Data').
               {fixed text of screen test, position of first character
                (row,col) followed by text to be displayed }

          field(test,command, 9, 13,  1).
          field(test,data,    11, 13, 60).
               {data-fields of screen test, name of field, position and
                length, empty fields are filled with a '.' }

          displayscreen(_screen) :-
            screen(_screen,_begin,_end), clear(_begin,_end),
            while( text(_screen,_lin,_col,_text),
                   put(_lin,_col,_text)          ),
            while( field(_screen,_name,_lin,_col,_len),
                   putd(_lin,_col,_len,'.')             ).

          while(_X, _Y) :- _X, _Y, fail.
          while(_X, _Y).
```

Note that 'clear', 'put' and 'putd' are builtins and implement some primitive actions for screenmanipulation.

Considering the call '?- displayscreen(test)' as the 'main program', our partial evaluation system transforms the program to the following equivalent program :

```
          displayscreen(test) :-
               clear(5, 15),
               (
                 put(5,15,'Screen A program X'),fail;
                 put(9, 5, 'Command'),fail;
                 put(11,5,'Data'),fail;
                 (
                   putd(9,13,1,'.'),fail;
                   putd(11,13,60,'.'),fail;
                   true
                 )
```

```
        ).
```

which involves only 7 logical inferences instead of 17 if we would use the original program.

Another example involving abstract data types could be an implementation of lists.

```
         empty(nil).
         cons(_e, _list, .(_e,_list)).
         append(_in, _out, _out) :- empty(_in).
         append(_in, _part, _out) :-
              cons(_e, _list1, _in),
              append(_list1, _part, _list2),
              cons(_e, _list2, _out).
```

Using the call '?- append(_X, _Y, _Z)' as target we obtain :

```
         append(nil,_X,_X).
         append(.(_e,_in), _part, .(_e,_out)) :-
                append(_in, _part, _out).
```

which is the program we would write when not using abstract data types (the variables are renamed for clarity). This implies that we can now safely use the concepts of abstract data types, without losing any efficiency during evaluation, thanks to the partial evaluation. A call to append using the first program would involve 3n+2 inferences (where n is the number of elements in the first inputlist), the transformed program only n+1.

Let us consider now an example which queries a simple database about family relationships :

```
         grandparent(_X,_Y) :- grandmother(_X,_Y).
         grandparent(_X,_Y) :- grandfather(_X,_Y).
         grandmother(_X,_Y) :- mother(_X,_Z), parent(_Z,_Y).
         grandfather(_X,_Y) :- father(_X,_Z), parent(_Z,_Y).
         parent(_X,_Y) :- mother(_X,_Y).
         parent(_X,_Y) :- father(_X,_Y).

         mother(Anna,Violette).
         mother(Violette,Jan).
         mother(Violette,Stan).
         mother(Henriette,Henry).
         mother(Henriette,Stanis).
```

Let us consider that the tuples for the father relation are not available at compile-time, but reside e.g. on an external database. Consider the call : '?- grandmother(_gm,Jan)'. The program resulting from partial evaluation :

```
         grandmother(Anna,Jan) :-
```

```
              (true ; father(Violette,Jan)).
          grandmother(Violette,Jan) :- father(Jan,Jan).
          grandmother(Violette,Jan) :- father(Stan,Jan).
          grandmother(Henriette,Jan) :- father(Henry,Jan).
          grandmother(Henriette,Jan) :- father(Stanis,Jan).
```

These results are accomplished by forward propagation of values through the program, opening calls, evaluating builtins whenever possible and applying some simplifications on the resulting clauses.

3. A META-INTERPRETER AS A SIMPLE PARTIAL EVALUATION SYSTEM.

The basis of our partial evaluation system is a Prolog meta-interpreter (i.e. a Prolog interpreter written in Prolog) :

```
          execute(_X) :- execute(_X,true).

          execute(true,true) :- !.
          execute((_X,true),_delay) :- !, execute(_X,_delay).
          execute((_X,_Y),_delay) :- !, execute(_X,(_Y,_delay)).
          execute((_X;_Y),_delay) :- !, execute(_X,_delay),
                                        execute(_Y,_delay).
          execute(_X,_delay)  :- !, clause(_X,_Y),execute(_Y,_delay).
          execute(_X,_delay)  :- !, call(_X),execute(_delay,true).
```

By changing the 'call(_X)' to 'write(_X)' we obtain a simple partial evaluation system which works only correctly for non-recursive, cut-free and deterministic rules.

4. AN EXTENSION FOR PROCEDURES CONSISTING OF MULTIPLE CLAUSES.

It is obvious that this system works not correctly for the following little example :

```
          acceptable(_class,_status) :- _class == 'High Quality'.
          acceptable(_class,_status) :- _status == 'Approved'.
```

where we assume a relation product which resides on an external database and which is not known at compile time :

```
          product(330,'Low Quality','Test').
          product(340,'High Quality','Approved').
          product(350,'Low Quality','Approved').
          product(360,'High Quality','Test').
          etc.
```

The program :

```
          program(_X,_Y) :- product(380,_X,_Y), acceptable(_X,_Y).
```

which would yield on the output file :

```
        product(380,_X,_Y), _X == 'High Quality', _Y == 'Approved'.
```

which is definitely not equivalent to the initial program. The problem rises whenever there are multiple definitions for a procedure, the meta-interpreter backtracks on 'clause(_X,_Y)' which it should do in the context of interpreting a program but obviously not in the context of partial evaluation. Therefore we changed the program : instead of simply writing down the calls to primitive actions we constitute a list of the sequence of calls encountered and write the completed list down at the end, on backtracking we generate a list for each solution which can be written down at the end. In the case of the example it would give :

```
        program(_X,_Y) :- product(380,_X,_Y), _X == 'High Quality'.
        program(_X,_Y) :- product(380,_X,_Y), _Y == 'Approved'.
```

Which reflects better what we want. However, when we add a primitive action, which has a side effect during execution, in front of the program, the transformed program is once again not equivalent to the initial one.

```
        program(_X,_Y) :- write(anything), product(380,_X,_Y),
                          acceptable(_X,_Y).
```

is transformed to :

```
        program(_X,_Y) :- write(anything), product(380,_X,_Y),
                          _X == 'High Quality'.
        program(_X,_Y) :- write(anything), product(380,_X,_Y),
                          _Y == 'Approved'.
```

The write would be executed twice here, in the initial program only once. In fact we want the following result :

```
        program(_X,_Y) :- write(anything), product(380,_X,_Y),
                          (_X == 'High Quality' ; _Y == 'Approved' ).
```

which can be achieved by putting the 'clause(_X,_Y)'-call in a 'setof' and treat the resulting list of bodies as an or-list. This or-list should recursively be transformed by the partial evaluation system. (The result of the partial evaluation is then and and-list where the elements can be calls to primitive actions or or-lists of and-lists).

It should be noted at this stage that this version of our partial evaluation system already takes in account the process of propagating the values through the program (by the unification at the moment of executing the clause(_X,_Y)) and the opening of the calls through the constitution of the list of consecutive calls.

The third technique, which consists of evaluating builtins whenever it is possible, is easily added to this partial evaluation system : every builtin which is known not to have a side-effect (e.g. arithmetic functions, etc.) are evaluated when possible, which instantiates possible new variables which can be propagated through the program otherwise the call is simply included in the output-list as we would do before.

This gives us the following program :

```
          execute((_X,_Y),_delay,_L,_M) :- !,
                    execute(_X,delay(_Y,_delay),_L,_M).
          execute((_X;_Y),_delay,_L,and(or(_res1,_res2),_L)) :-
                    execute(_X,_delay,nil,_res1),
                    execute(_Y,_delay,nil,_res2).
          execute(_X,_delay,_L,_M) :- builtinpred(_X,_L,_N), !,
                    executedelay(_delay,_N,_M).
          execute(_X,_delay,_L,and(_or,_L)):-
                    setof(_B,allclauses(_X,_delay,_B),_M),
                    orlist(_M,_or).
          execute(nil,_delay,_part,_res) :-
                    executedelay(_delay,_part,_res).

          {meaning of the procedures and parameters :
          execute :  1st par. is calls to be evaluated
                     2nd par. is list of calls to be evaluated later
                     3rd par. is partial result
                     4th par. resulting list (output)
          setof is a builtin which constitutes a list _M of all _B
           which are a solution of allclauses(_X,_delay,_B)
          orlist converts a 'standard' list to an or-list }

          executedelay(nil,_res,_res).
          executedelay(delay(_X,_delay),_part,_res) :-
                    execute(_X,_delay,_part,_res).

          allclauses(_X,_delay,_B) :- clause(_X,_Y),
                                      execute(_Y,_delay,nil,_B).

          {note here the recursive call to 'execute' to evaluate the
           bodies of the procedures, before inclusion in the list}

          builtinpred(_X,_L,and(_X,_L)) :- builtin(_X), sideef(_X), !.
          builtinpred(_X,_L,_L)         :- builtin(_X), call(_X), !.
          builtinpred(_X,_L,and(_X,_L)) :- builtin(_X).
```

at which we should add some 'declarations' about the builtin predicates of the system :

```
          builtin(write(_X)).
          sideef(write(_X)).
          builtin(nl).
```

```
        sideef(nl).
        builtin(_X is _Y).
        builtin(_X = _Y).
        etc.
```

The program to be optimised can be read in core and the invocation of the partial evaluation could be as follows :

```
        ?- execute(program,nil,nil,_OL), prettyprint(_OL).
```

{where prettyprint(_X) is a procedure to output the result in a human readable way}

This constitutes the basis of our partial evaluation system which we will elaborate further in the sequel of this paper.

5. FROM THEORY TO PRACTICE : SOME AMELIORATIONS.

The first program we tried to optimise with this partial evaluation system was the partial evaluation system itself, which did not work as we liked to. There were several reasons for this which we explore in this section and whose investigation resulted in a final system that copes with full Prolog programs.

5.1 RECURSION.

The system obviously gets in a loop each time it encounters a recursive rule. In the literature concerning compilation techniques for queries in recursive first-order databases (e.g. [6]) one tries to solve this by compiling recursive rules into iterative programs, which does not suit our approach of compiling into Prolog. We propose (and implemented) a simple but usable solution.

During the process of partial evaluation we keep track of the procedures we currently are working on (e.g. in a list), when we encounter a recursive call (which is thus an element of this list), we suspend the process of partial evaluation on this call, but transfer it as it was to the output list. In order to obtain an executable program this call to a recursive rule should be queued. After ending the partial evaluation of the input program, the system starts evaluating the queued calls to recursive rules. The goal is to obtain 'execution plans' for procedures which consist of a sequence of builtin-invocations and calls to recursive procedures. For each recursive procedure used we generate an 'execution plan'.

An example :

```
        program(_X) :- fac(_X).

        fac(_X) :- _X == 0, write(1).
        fac(_X)  :-  _X  \==  0,  write(_X),  write('*'), _Y is _X-1,
fac(_Y).

          {generation of a product that calculates n! }
```

The solution we propose generates :

```
          program(_X) :- (_X == 0, write(1) ;
               _X \== 0, write(_X), write('*'), _Y is _X-1, fac(_Y) ).

          fac(_X) :- (_X == 0, write(1) ;
               _X \== 0, write(_X), write('*'), _Y is _X-1, fac(_Y) ).
```

which can be executed by the Prolog system and is equivalent to the initial program. One can verify that an execution of the transformed program needs less logical inferences than the initial program.

Note that this approach is too conservative in certain cases : a call like '?- fac(28)' can be transformed savely with the initial system to '?- write(28), write('*'), write(27), write('*'), ...' because the only parameter is instantiated and the procedure 'fac(_X)' is written correctly. The problem however rises when parameters are only partialy instantiated, in this case a decision whether to delay partial evaluation or not, should be based upon whether the parameters are sufficiently instantiated to prevent infinite recursion. This could be accomplished when one adds declarations about legal uses of procedures (all valid input-output patterns) in combination with a dataflow-analysis which checks the consistency of the different declarations with the program (see [1] and [11] for more details). Some related work has been presented in [9].

5.2 CUT.

The simple meta-interpreter which we introduced in section 2 does not work correctly for rules with cuts. So we were not surprised that the transformations of programs containing cuts did not work correctly. A cut has a very specific impact on the control of a very specific part of the program (i.e. only the subgoal where it appears within), this notion of context should be preserved in the generated program. For this reason we introduce a special control structure for keeping track of this context.

An example :

```
          program :- pos(_toy,_X), fac(_X).
          fac(_X) :- _X == 0, !, write(1).
          fac(_X) :- write(_X), _Y is _X-1, write('*'), fac(_Y).
```

with a relation 'pos' on an external database.

```
          pos(cubblestone,6).
          pos(roulette,36).
          etc.
```

The following program is obviously not equivalent :

```
         program :- pos(_toy,_X), (_X == 0, !, write(1) ;
                    write(_X), _Y is _X-1, write('*'), fac(_Y)).
```

The cut here (as we take the definition of the cut in Cprolog) has an impact on a too great part of the program, the context should be restricted to the or-clause. This can be expressed introducing an appropriate marking and a modified cut referring to this marking :

```
         program :- pos(_toy,_X), mark(1), (_X == 0, !(1), write(1) ;
                    write(_X), _Y is _X-1, write('*'), fac(_Y)).
```

This program needs a modified Prolog system which handles those special mark- and cutpredicates. A mark(n) always succeeds, on reaching a cut(n) the modified interpreter removes all backtrackpoints backwards till the mark(n). It is very easy to make this little change in a Prolog interpreter or to write a meta-interpreter which copes with this special control structure. A similar control structure has been proposed in [3].

5.3 BACKWARD UNIFICATION.

One technique of partial evaluation consists of propagating values forward through the program, which is accomplished in our system by the unification on calling 'clause(_X,_Y)'. This unification however does more than instantiating input parameters of the called procedure, it also eventually gives back values for the output parameters which in certain cases causes problems.

An example that illustrates these problems :

```
         program(_X) :- p(_X).
         p(a) :- write(a).
         p(b) :- write(b).
         p(_X) :- write(ok).
```

The partial evaluation system generates :

```
         program(a) :- (write(a) ; write(ok)).
         program(b) :- write(b).
```

because of the backward unification the system generates 2 solutions for p(_X) one with _X='a' and one with _X='b', while the solution that _X remains a free variable is lost. Note that in such cases the setof-statement backtracks and generates multiple solutions.

There are two ways to solve this problem. The first one is very simple but restricts the impact of the partial evaluation considerably. The second is somewhat more difficult to implement but gives the best results.

The first solution consists of making a copy of the call and partially evaluate it, the system continues by checking whether the unification involves backward assignments, if not the copy of the call is unified with the initial call, if so it delays the call as it would do with recursive ones. In the example above the program would stay unchanged and thus correct.

Note that this restriction is too conservative : when the called procedure is deterministic (i.e. it has only one solution), the backward unification is save. In general one can not decide at compile time whether a procedure is deterministic or not (e.g. when the user is allowed to add assertions). When using a modular program structure however, which protects rules against illegal userinteraction, one could include this check for determinism in the partial evaluation system.

A program like :

```
          program(_X) :- p(_X).
          p(a).
```

could then safely be transformed into :

```
          program(a).
```

But as we said before, this solution has a too restrictive effect on the impact of the partial evaluation process, while the percentage of backward unification in a typical Prolog program is usualy very high.

The second solution starts from a number of copies of the initial call, one for each possible solution, the solutions are put in an orlist. Instead of unifying the seperate calls with the initial one, only the forward unification is executed, for each backward unification an explicit assignment is added to the respective solution. A little example illustrates :

```
          program(_X) :- ( _X=a, write(a) ;
                           _X=b, write(b) ;
                           _X=_Y, write(ok) ).
```

This solution can then further be simplified into :

```
          program(a) :- write(a).
          program(b) :- write(b).
          program(_X) :- write(ok).
```

We do not intend to go in details concerning the implementation of the final system which copes with recursion, cut and backward unification. The final system consists of nearly 200 lines of Prolog code, from which 25 lines implement the 'pretty printing', 20 lines the simplication of the solutions, 20 extra lines were needed for coping with recursion, 15 lines for the cut and some 35 lines for backward unification. Some extra code was needed to cope with some special features of Prolog like meta-calls, not, ...

Tests with 'real-life' Prolog programs gave the same results as described in [8]. It is obvious that 'dead' code is automatically eliminated. The number of procedures decreases significantly : only the 'main program' is preserved together with execution plans for all used recursive rules. The volume of the code however can grow considerably, depending on the nature of the program. A program using an in-core database e.g. can give rise to a combinatorial explosion of code. However the number of logical inferences required to execute the transformed program is always less than those required to execute the initial program, with a factor in the range of 1.5 to 5 or more. The gain achieved with the partial evaluation is the greatest when the programs were written using structured programming techniques as modular programming, abstract data types, etc., and the performance of the transformed programs is comparable (if not better) with the performance of manually optimised programs. In other words : the partial evaluation system relieves the programmer from concerns about optimisation and permits him to use programming methodologies who are closer related to logic programming and to focus himself on issues of efficient problem solving.

6. APPLICATION OF PARTIAL EVALUATION ON QUERY OPTIMISATION.

The objective here is to couple a Prolog system to a database system in a way that seems to be a good compromise between the compiled approach as described in [2] and the interpretive approach first implemented in [10]. By adding some declarations concerning the tuples stored in the extensional database (edb) the former method generates, by delaying edb-calls, a conjunction of calls to the edb, that can be solved by the database system (this method only works for non-recursive, cut-free, builtin-free rules). The latter method requires a little change in the Prolog interpreter in the sense that each time a call to a relation is encountered which is stored on the database, the call is transmitted to the DB-system for execution, the solutions are stored on a stack and consumed one by one by the normal backtracking of the Prolog system.

Our approach consists of extending the partial evaluation system as to treat calls to the edb as not yet evaluable builtins. This implies that the transformed program consists of conjunctions of calls to the edb intermixed with calls to builtins and recursive rules, which themselves have the same structure. The corresponding Prolog system operates in the same way as the one in the interpretive approach, but instead of transmitting calls to single relations to the edb, it transmits conjunctions of calls, which can be the subject of some optimisation at the level of the database system (or possibly at compile time), the results of the query are again stored on a stack and consumed one by one through backtracking.

The advantage of this approach is that the transformed program remains a Prolog program (Prolog extended with mark(n) and cut(n)), that the system copes with recursive queries and that calls to the external database are clustered together, which enables some query optimisation.

Investigation should point out whether the optimisation on these conjunctions of edb-calls should be done at compile time, (as in [12], using statistical information from the database system or as in [5] or [7] where semantical information is used for semantic query optimisation) or whether the optimisation should be left over to the available database system.

7. CONCLUSION.

We presented some results of the investigation to applying the technique of partial evaluation to Prolog. The main goal was to obtain a tool that could relieve the programmer from concerns of optimisation issues. The partial evaluation system, which is based on a simple Prolog meta-interpreter, is shown to work for full Prolog program. The Prolog programs are transformed in equivalent (partially evaluated) MC-Prolog programs, where MC-Prolog is a Prolog extended with mark(n) and cut(n) builtins. It is shown how partial evaluation can be used as a compromise between the compiled and interpreted approach in linking Prolog to database systems.

8. ACKNOWLEDGEMENTS.

This work has been done partly under the contract KBAR/SOFT/1, which is a grant of the DPWB (Diensten van de programmatie van het wetenschapsbeleid). The application of partial evaluation on the integration of Prolog and databases has been investigated as part of the ESPRIT pilot project 3.3, titled "A logic oriented approach to knowledge and databases supporting natural user interaction".

9. REFERENCES.

[1] Bruynooghe Maurice, Adding redundancy to obtain more reliable and readable Prolog programs, Proc. of the 1st Int. Logic Programming Conf., Marseille 1982, pp. 129-133.
[2] Chakravarty, U.S., Minker, J. and Tran, D., Interfacing predicate logic languages and relational databases, Proc. of the 1st Int. Logic Programming Conf., Marseille 1982, pp. 91-98.
[3] Chikayama T., ESP as a preliminary kernel language of fifth generation computers, ICOT Research Center Technical Report, 1983.
[4] Gallaire H. and Minker J., Logic and Data Bases, Plenum Press, New York 1978.

[5] Hammer M. and Zdonik S., Knowledge based query processing, Proc. 6th Int. Conf. on Very Large Database, Montreal 1980, pp. 137-147.
[6] Henschen L.J. and Naqvi S.A., On compiling queries in recursive first-order databases, Journal of the ACM, Vol 31 Number 1, Jan 1984.
[7] King J., Quist : a system for semantic query optimisation in relational databases, Proc. 7th Int. Conf. on Very Large Databases, Cannes 1981, pp. 510-517.
[8] Komorowski Henryk Jan, A specification of an abstract Prolog machine and its application to partial evaluation, Linkoping Studies in Science and Technology Dissertations No 69, Linkoping University 1981.
[9] Naish L., Automatic generation of control for logic programs, Technical Report 83/6, Department of Computer Science, The University of Melbourne 1983.
[10] Venken Raf, A simple relational database as an extension for Prolog (in Dutch), undergraduate dissertation, Katholieke Universiteit Leuven 1981.
[11] Venken Raf and Bruynooghe Maurice, Prolog as a language for prototyping, Proc. of Working Conference on Prototyping, Namur 1983.
[12] Warren, D.H.D., Efficient processing of interactive relational database queries expressed in logic, Proc. 7th Int. Conf. on very Large Databases, Cannes 1981.