An inference engine for function free logic programs

Stefan Böttcher
IBM Deutschland GmbH
Scientific Center
Institute for Knowledge Based Systems
P.O.Box 80 08 80
D – 7000 Stuttgart 80
West Germany *

Abstract

This paper presents the evaluation of function free logic programs by the PROTOS-L system. PROTOS-L is a logic programming language which embeds a module concept, provides read access to external databases and combines order-sorted types with polymorphism. From this viewpoint, PROTOS-L is similar to DATALOG embedded in a typed logic programming language.

The presentation focusses on the evaluation of queries to function free logic programs within the PROTOS-L system. We use a small fragment of a travel information system in order to illustrate the evaluation of queries to function free PROTOS-L logic programs.

1 Introduction

This section summarizes some of the basic language features of PROTOS-L which have already been described elsewhere. It furthermore outlines the example program which will be used throughout this paper.

1.1 An overview of PROTOS-L

PROTOS-L is a logic programming language extended by the following features: access to relational databases, a module concept, a type concept including subtypes and polymorphism. Furthermore, PROTOS-L supports the programming of deductive databases.

The type system of PROTOS-L which is described e.g. in [Beierle and Böttcher, 1989] supports subtypes and polymorphism, i.e. type variables. It is derived from the type system of TEL [Smolka, 1988]. PROTOS-L types are not only used by the compiler for type checking, but also by the PROTOS Abstract Machine (PAM) [Semle, 1989], [Böttcher and Beierle, 1989] for computations on order-sorted types in order to reduce the search space.

*The research reported here has been carried out within the international EUREKA project PROTOS (EU56): Prolog Tools for Building Expert Systems. Project partners are BIM, IBM Stuttgart, Sandoz AG, Schweizerische Bankgesellschaft, University of Dortmund and University of Oldenburg.

The module concept of PROTOS-L [Beierle, 1989], [Böttcher, 1990a] integrates ideas from the module concepts of Modula-2 [Wirth, 1983], TEL [Smolka, 1988] and DBPL [Eckhardt *et al.*, 1985], [Böttcher, 1989]. However, PROTOS-L offers two kinds of module bodies: program bodies and database bodies. Like other implementation details, at the interface of a module it is not visible, whether the corresponding module body is a program body or a database body. *Program bodies* support logic programming with backtracking. *Database bodies* support access to relational databases and the implementation of deductive databases, i.e. the implementation of function free logic programs accessing these databases. The language of PROTOS-L database bodies has the power of DATALOG, i.e. every DATALOG program can be expressed in a PROTOS-L database body.

Database bodies are evaluated by an efficient set-oriented query evaluation strategy which is described in section 3, whereas program bodies are evaluated by backtracking. Note that it is the decision of the PROTOS-L programmer, which evaluation strategy he wants to use for function free logic programs: Whenever the programmer wants a predicate to be evaluated by backtracking he has to implement it in a program body. However, if he prefers set-oriented retrieval he simply implements his function free rules in a database body.

1.2 An example of a PROTOS-L function free logic program

Before we describe the details of the set-oriented evaluation of function free logic programs in the PROTOS-L system, we give a small example for a function free logic program written in PROTOS-L. The example program is a function free part of a travel information system which has been described in more detail in [Böttcher, 1990a]. The example program has been slightly modified in order to simplify a performance comparison of the evaluation strategy described in this paper with the performance of Quintus Prolog.

The example program fragment of our travel information system is shown in figure 1. It computes the departure time, the arrival time and the number of intermediate stops for `flight_connections` from a large set of given `direct_flights`. The first rule

```
database_body flight_info using flight_DB .

 rel   flight_connections :  ?int x  ?string x  ?string x  ?int x     ?int .
 %                           stops   from       to         departure  arrival
 % Which flight connections are possible for which time ?

 flight_connections( 0 , From , To , Departure , Arrival )
 <--  direct_flight( From , To , Departure , Arrival ) .

 flight_connections( Stops+1 , From , To , Departure , Arrival )
 <--  direct_flight( From , Change , Departure , Arrive_at_Change )
   &  flight_connections( Stops, Change, To, Depart_from_Change, Arrival )
   &  Arrive_at_Change + 100 < Depart_from_Change .


 rel    direct_flights :  ?string x  ?string x  ?int x     ?int     .
 %                        from       to         departure  arrival
 % Which direct flights are scheduled for which time ?

 dbrel  direct_flights    is
        Flight_Rel( FROM , TO , DEPARTURE , ARRIVAL ) .

endmodule.
```

Figure 1: A function free logic program computing flight connections from acyclic data for direct flights

states that a non-stop flight connection is a direct flight. The second rule computes those n-stop flight connections from direct flights and (n-1)-stop connections that leave enough time to change the airplane (one hour in our example).

The program fragment shown in figure 1 is a database body, i.e. the predicates defined in the database body are evaluated by the efficient set-oriented evaluation strategy described in section 3 and some facts for predicates are taken from a database relation. The former is the case for the predicate `flight_connections`. The latter is the case for the relation `direct_flights` in our example, the facts of which are taken from the database relation `Flight_Rel` of the database `flight_DB`.[1]

Apart from the travel information system, PROTOS-L is currently used to reimplement a part of a chemical production planning system [Böttcher, 1990c]. PROTOS-L has been implemented at IBM Stuttgart on an IBM-RT 6150 workstation.

2 Why we do not use the deduction strategy of Prolog for function free logic programs

This section summarizes the advantages of bottom-up and top-down query evaluation. It focusses on the reasons why the PROTOS-L system contains a special purpose theorem prover for function free logic programs.

Although the implementation of the PROTOS-L system embeds a database system and allows the retrieval of facts from database relations for the evaluation of function free logic programs, most of the following arguments and, even more importantly, the algorithms described in the next section do not require that facts are physically stored in a database. The improvements of our algorithm are applicable as well, if the facts are coded in logic programs and stored together with the other program code in main memory.

The Prolog deduction strategy is depth first, left to right with backtracking. It reduces the search space, because it works goal-oriented, i.e. it only derives subgoals which might contribute to the answer of a query. This goal-oriented deduction has been combined with bottom-up (forward) deduction, e.g. in the magic set approach [Bancilhon and Ramakrishnan, 1986]. The PROTOS-L inference engine for function free logic programs also uses this goal-oriented deduction, and is therefore superior to naive or semi-naive inference engines [Bancilhon and Ramakrishnan, 1986].

The following states the reasons why we do not use the Prolog evaluation strategy for deductive databases in the PROTOS-L system:

- The Prolog evaluation strategy may lead to infinite left recursion, although there exists a proof for a goal. We want to have a proof procedure which always terminates. There is no termination problem for bottom-up deduction, because for DATALOG programs all rules are function free.
- The Prolog evaluation strategy does a lot of redundant computation. For example solutions to a predicate `q(X,Z)` are recomputed for every occurrence of a goal `q(X,Z)` in a logic program.

  The idea of lemma generation is that all intermediate results are stored for further usage. The

[1] The database relation `Flight_Rel` of the database `flight_DB` must have at least the four attributes: `FROM` and `TO` of type string, and `DEPARTURE` and `ARRIVAL` of type integer.

Prolog evaluation strategy does not use lemma generation. This leads to a large number of redundant computations in typical DATALOG programs.

- The Prolog evaluation strategy leads to an expensive computation of large joins. Consider the following example (which is written in PROTOS-L):

  ```
  s(X,Z) <-- r(X,Y) & r(Y,Z) .
  ```

  Assume that the relation r contains 1000 ground facts (or the goal r(X,Y) yields 1000 ground answers). In the worst case, the Prolog strategy proves 1000 goals r(X,Y) and 1000 × 1000 goals r(Y,Z) in order to compute the answers to s(X,Z). Here, it would be much faster to use efficient join algorithms, e.g. the sort-merge algorithm [Ullman, 1982].

The proof strategy of the PROTOS-L deductive database system contains the following improvements in order to avoid the disadvantages of the Prolog proof strategy.

- There is no infinite left recursion possible with the PROTOS-L proof strategy.
- The PROTOS-L proof strategy uses lemma generation for every predicate.
- PROTOS-L proofs use a fast computation of large joins.

Remember that PROTOS-L does not require to evaluate every predicate in a set-oriented way but only those predicates for which the PROTOS-L programmer requires a set-oriented evaluation (by implementing them in a database body). In order to support evaluation strategies, the PROTOS-L system embeds its special purpose theorem prover in an extension of the Warren Abstract Machine [Warren, 1983].

3 The process model

This section describes the inference engine of the PROTOS-L deductive database system. A more comprehensive description of the algorithms is given in [Meyer, 1989] the basic idea of which is summarized in this section.

3.1 The inference engine of the process model

A PROTOS-L deductive database contains only function free rules which additionally have the property that after every bottom-up (forward chaining) application of such a rule every variable occurring in the head is bound.

The PROTOS-L deductive database inference engine uses an approach which combines the QoSaQ approach [Vieille, 1988] with ideas of [Hulin, 1989]. The basic idea of the PROTOS-L deductive database inference engine is the process model. Every predicate (occurring in a PROTOS-L deductive database) is implemented by its own *process*. The rules implementing a process and the goals occuring in these rules are considered to be part of the process. It is easier to think of independent processes in order to understand the process model. Nevertheless, the whole computation is implemented as a single operating system process within the PROTOS-L system because this reduces the communication costs and it makes the evaluation much faster.

Every process has two kinds of memories:

- one memory in order to store all received queries together with their environments,
- one memory in order to store all received answers.

For example, we look at two predicates p and q where q is defined by

```
q(X,Z) <-- ... & p(X,Y,Z) & ... .
```

The memories of a process computing the answers to a predicate p could contain the following queries and answers.

	queries	and	answers
(1)	?- p(1,X,Y)		p(1,2,3)
(2)	?- p(1,2,Z)		p(1,4,5)
(3)	?- p(1,W,W)		...

The processes are coordinated by a global scheduler. Every process can perform three kinds of activities:

- It can submit a subquery. For example, the process q submits a subquery to the process p.
- It can submit an answer. For example, the process p submits answers to the process q. This reactivates process q, if q is idle.
- It signals to the global scheduler that it is *idle* when is has submitted all of its subqueries and all of its answers. For example, the process p can signal that it is idle. Note however that the process p is reactivated, if it receives a query from the process q.

Query processing is completed when all processes signal to the global scheduler that they are idle. A process signals that it is idle, when it has computed all answers to all subqueries it has received up to now and it has submitted every answer to all queries which can be solved by the answer. But the process removes the idle signal, when it has received some queries which have not been processed completely.

Every process decides on its own, which subqueries and which answers it wants to submit first. The global scheduler assigns priorities to the processes thereby controlling the evaluation strategy, e.g. it can assign higher priorities to processes at the bottom in order to perform a bottom-up evaluation.

3.2 Lemma generation within the process model

The basic idea of lemma generation is to reuse answers to intermediate queries (lemmas), instead of recomputing them. The process model supports lemma generation as follows. No process computes answers to any query submitted to it twice. Even more, whenever a query is subsumed by some other query which is already in the query table of the process, then the process will not submit further subqueries for the subsumed query. The reason is that the subsumed query can be answered by the facts solving the subsuming query, but it can not get any other answer. Instead of processing subqueries twice, the stored answers of a process are submitted to all goals they satisfy. That is how PROTOS-L supports lemma generation for every predicate occurring in a database body.

For example, look at the query table of the process for some predicate p which is shown above. The queries (2) and (3) are subsumed by the query (1) and therefore they do not produce any new subqueries. The answers to any query for p are used not only for query (1) but are reused to answer the queries (2) and (3) whenever this is correct.

It is not necessary to compute completely the answers to some subsuming query, say query (1), before they can be reused for a subsumed query, say query (2). Instead, all answers found in the answer table of *p* can be submitted to any query to *p* they solve without delay.

The second part of the lemma generation is the following. Received answers for a predicate p will be matched with all queries to p stored in the query table of p. Whenever such a new answer to p matches a goal for p occurring in a rule of process q, the then answer of p is submitted to the process q. Together with the answer q receives from p the environment with which it called the goal to p because we store these environment in the query table of p, i.e. the environment is stored local to p and not local to q.

3.3 Alternatives and special cases of the process model

The process model provides several degrees of freedom:

- The priorities can be assigned arbitrarily to the processes. This leads to different evaluation strategies.
- Each process can perform an arbitrary number of actions before it returns the control to the scheduler. For example, it can submit only one subquery or one answer at a time or it can submit an arbitrary other number of subqueries or answers at a time.
- Join computation by side-way information passing could be done tuple-oriented or set-oriented.

The process model can emulate other strategies for recursive query evaluation by selecting adaptive alternatives for the parameters. For example, semi-naive bottom-up evaluation of function free logic programs is achieved by assigning higher priorities to those predicates which are closer to the bottom, by using lemma generation and by computing all answers to a query at a time.

Furthermore, the depth-first strategy of Prolog is similar to a process model not using lemma generation where higher priorities are assigned to those predicates which are closer to the query, and where processes submit further subqueries for one answer to a goal at a time.

4 Implementation of the process model within the PROTOS-L system

4.1 Inference engines of the PROTOS-L system

The PROTOS-L inference system consists of two inference engines which are implemented as abstract machines. The upper inference engine (called the PROTOS Abstract Machine (PAM)) is an extension of the Warren Abstract Machine by types and polymorphism. This upper inference engine uses backtracking. The lower inference engine (called deductive database inference engine (DDBIE)) is a special purpose inference engine for function free logic programs implementing the process model. Its proof technique has been described in section 3 and some performance results are described in section 4.3. The performance results of section 4.3 show that this inference engine is adapted for proofs which contain large sets of ground facts. The integration of both inference engines in described in [Böttcher, 1990b].

Alltogether, PROTOS-L provides backtracking on top of set-oriented proofs. Note that it is the decision of the PROTOS-L programmer, in which cases he assumes that solutions are easy to find and therefore prefers backtracking, and in which cases he prefers set-oriented retrieval of facts, because he assumes that a large search space has to be searched in order to find a solution. Remember, whenever the PROTOS-L programmer prefers tuple-oriented unification with backtracking he programs his rules in program bodies, because program bodies are compiled into code which at run time is evaluated by the upper inference engine. Otherwise, if he prefers set-oriented retrieval he programs his rules in database bodies. Database bodies are compiled into code which is evaluated by the lower inference engine.

4.2 Implementation of the process model

Each PROTOS-L database body is compiled into code which at run time constructs a query graph from the function free logic program contained in the database body. This query graph is instantiated with the variable bindings of a goal given by the PAM. In the following we summarize, how solutions to a given goal predicate are computed recursively by submitting subqueries to those subgoal predicates which occur in the rules defining the goal predicate. The query graph (which is similar to a rule goal graph) contains information about the sequence of goals in a rule. For example, look at the rule for process q given in section 3.2. This sequence is used in order to determine to which further subgoal a subquery has to be submitted when an answer to a given goal (p) is received by the process q. The variable bindings for the next subgoal are computed from the received answer for p and from the environment stored in the query table of p.

Although our implementation allows the retrieval of facts from a database system, the process model does not require that facts are stored physically in a database. Instead, facts may as well be taken from program code and stored together with the program code in main memory.

The process model requires the assignment of two kinds of memories to each process: a memory for queries and their environments and a memory for answers. Our implementation uses main memory tables (organized as lists of entries) in order to implement these memories, i.e. every process is implemented by appropriate operations on its main memory tables.

4.3 Performance results

The travel information system, part of which is described above, has been implemented in PROTOS-L and in Quintus Prolog (version 2.4, using the compiler) in order to get some performance results comparing the process model with the evaluation strategy of Prolog. Both implementations have been run on an IBM-RT 6150 workstation.

In the PROTOS-L version, the database contained 100 records for direct flight connections. In the Quintus Prolog version all facts where coded in the Prolog program. The query was to compute whether or not there exists a flight connection between two cities (the data was chosen in such a way that the longest path between the two cities had the length n). For n=4 we got the result that the computation of the path by the DDBIE is more than 6 times faster than the computation of the path by Quintus Prolog (although the DDBIE had to read all facts from an SQL/RT database system, whereas in the Quintus Prolog implementation the facts where coded in the Prolog program).

In order to generalize this performance result, we made further performance evaluations with the following result: The more facts are derived for a predicate, the more superior the DDBIE is compared to the Prolog evaluation strategy. Furthermore, the deeper the proof tree is, the more superior the DDBIE is compared to the Prolog evaluation strategy.

5 Summary and conclusion

PROTOS-L provides a module concept which supports transparent database access by hiding the implementation of predicates from the user of a module. Additionally, the programming of deductive databases is supported by database bodies.

PROTOS-L offers set-oriented evaluation strategies for rules contained in database bodies and backtracking for rules contained in program bodies. Nevertheless, the PROTOS-L programmer has to learn only one single language, because program body rules and database views are expressed in the same way. This avoids the mismatch of other integrations of database languages into programming languages.

Rules contained in database bodies are evaluated by the process model in a set-oriented way and incrementally on demand. Query evaluation by the process model is superior to query evaluation by the Prolog evaluation strategy, if a large set of facts (in the example 100 or more) of the same relation is given in the search problem and the depth of the proof tree is large enough (in the example greater than or equal to 4).

However, if the search space for a given goal is smaller, then backtracking is available to the PROTOS-L programmer, because the PROTOS-L system supports both set-oriented query evaluation and backtracking. Therefore, the PROTOS-L system which integrates both evaluation strategies supports adaptive evaluation of small search spaces as well as of large search spaces.

References

[Bancilhon and Ramakrishnan, 1986] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington D.C., 1986.

[Beierle, 1989] C. Beierle. Types, modules and databases in the logic programming language PROTOS-L. In K. H. Bläsius, U. Hedtstück, and C.-R. Rollinger, editors, *Sorts and Types for Artificial Intelligence*, Springer-Verlag, Berlin, Heidelberg, New York, 1989. (to appear).

[Beierle and Böttcher, 1989] C. Beierle and S. Böttcher. PROTOS-L: Towards a knowledge base programming language. In *Proceedings 3. GI-Kongreß Wissensbasierte Systeme, Informatik Fachberichte*, Springer-Verlag, Berlin, Heidelberg, 1989.

[Böttcher, 1989] S. Böttcher. *Prädikative Selektion als Grundlage für Transaktionssynchronisation und Datenintegrität*. PhD thesis, FB Informatik, Univ. Frankfurt, 1989.

[Böttcher, 1990a] S. Böttcher. Development and programming of deductive databases with PROTOS-L. In L. Belady, editor, *Proc. 2^{nd} International Conference on Software Engineering and Knowledge Engineering*, Skokie, Illinois, USA, 1990.

[Böttcher, 1990b] S. Böttcher. Integrating a deductive database system with a Warren Abstract Machine. In N. Cercone, F. Gardin, and G. Valle, editors, *Proc. Computational Intelligence III - The International Conference on Computational Intelligence 90*, Milan, Italy, 1990. (to appear).

[Böttcher, 1990c] S. Böttcher. A tool kit for knowledge based production planning systems. In M. Tjoa, editor, *Proc. International Conference on Data Base and Expert System Applications*, Springer-Verlag, Vienna, Austria, 1990. (to appear).

[Böttcher and Beierle, 1989] S. Böttcher and C. Beierle. Data base support for the PROTOS-

L system. *Microprocessing and Microcomputing*, 27(1–5):25–30, August 1989.

[Eckhardt *et al.*, 1985] H. Eckhardt, J. Edelmann, J. Koch, M. Mall, and J. W. Schmidt. *Draft Report on the Database Programming Language DBPL*. DBPL-Memo 091-85, Univ. Frankfurt, 1985.

[Hulin, 1989] G. Hulin. Parallel processing of recursive queries in distributed architectures. In *Proceedings of the 15th International Conference on Very Large Data Bases*, Amsterdam, 1989.

[Meyer, 1989] G. Meyer. *Regelauswertung auf Datenbanken im Rahmen des PROTOS-L-Systems*. Diplomarbeit Nr. 630, Universität Stuttgart, December 1989.

[Semle, 1989] H. Semle. *Erweiterung einer abstrakten Maschine für ordnungssortiertes Prolog um die Behandlung polymorpher Sorten*. Diplomarbeit Nr. 583, Universität Stuttgart und IBM Deutschland GmbH, Stuttgart, April 1989.

[Smolka, 1988] G. Smolka. *TEL (Version 0.9), Report and User Manual*. SEKI-Report SR 87-17, FB Informatik, Univ. Kaiserslautern, 1988.

[Ullman, 1982] J.D. Ullman. *Principles of Database Systems*. Computer Science Press, Rockville, 1982.

[Vieille, 1988] L. Vieille. From QSQ towards QoSaQ: global optimization of recursive queries. In *Proceedings of the 2nd International Conference on Expert Database Systems*, Virginia, 1988.

[Warren, 1983] D. Warren. *An Abstract PROLOG Instruction Set*. Technical Report 309, SRI, 1983.

[Wirth, 1983] N. Wirth. *Programming in Modula-2*. Springer, Berlin, Heidelberg, New York, 1983.