A semantics for the integration of database modifications and transaction brackets into a logic programming language

Stefan Böttcher
IBM Deutschland GmbH
Scientific Center
Institute for Knowledge Based Systems
P.O.Box 80 08 80
D - 7000 Stuttgart 80
West Germany [1]

Abstract

The integration of logic programming and databases has up to now focussed on read access of logic programs to external databases storing permanent data. However, the integration of write operations modifying existing databases would allow to use logic programming languages in a much larger field of applications. Therefore, it is important that the logic programming language not only embeds modification operations on existing databases, but also embeds transactions in order to preserve the correctness of the modified database.

This paper describes an integration of database modifications and transactions into a logic programming language evaluated by a depth first left to right strategy with backtracking. We propose a semantics of insert and delete operations and outline why these operations are defined different from assert and retract in Prolog. Furthermore, we propose a semantics for transaction brackets and describe how these operations differ from begin_transaction and end_transaction statements in database programming languages.

[1] The research reported here has been carried out within the international EUREKA project PROTOS (EU56): Prolog Tools for Building Expert Systems.

1 Introduction

During the last years, the integration of logic programming and databases has become an increasingly important research area, because the integration enables logic programs to use the knowledge stored in existing databases. However, in order to fully support high-level database access from logic programs, the integration of database modifications has been considered to be an important extension of logic programming (e.g. [Fagin *et al.*, 1986], [Manchanda and Warren, 1988], [Wilkins, 1986]). In contrast to these approaches we argue that correctness of database updates additionally requires to integrate a transaction concept and that this may lead to a different judgement about update semantics.

While the integration of database modifications and transactions into procedural programming languages as e.g. Pascal or Modula-2 is well understood [Schmidt, 1977], [Schmidt *et al.*, 1988], the integration of these concepts into logic programming languages leads to the following problem: How shall we integrate database modifications and a transaction concept into the backtracking evaluation strategy used for logic programming languages ?

The problem can be devided into two subproblems: first, to find a clean integration of database modifications avoiding "dirty side-effects", and second, to integrate a transaction concept into backtracking such that atomicity and persistence of transaction executions are not violated, when several transactions are executed by one single logic program.

In this paper we describe a proposal to solve this problem based on the logic programming language PROTOS-L, which is currently developed at IBM Scientific Center in Stuttgart as part of the EUREKA project PROTOS (EU 56). PROTOS-L [Beierle, 1989], [Beierle and Böttcher, 1989] is a logic programming language, providing access to external databases, a polymorphic order-sorted type concept, and a module concept similar to that of Modula-2 [Wirth, 1983]. A compiler and an abstract machine[2] for PROTOS-L have been implemented on the IBM-RT 6150 workstation [Semle, 1989].

PROTOS-L is currently used to reimplement a part of an algorithm which is used in a prototype of a chemical production planning system for Sandoz AG. This production planning system uses large sets of heuristic rules coded in logic and needs access to planning data which is stored in a relational database system. Therefore it requires to embed database access into a logic programming language.

The next section summarizes the requirements, while the third section describes a proposal for the integration of database modification operations and a transaction concept into the logic programming language PROTOS-L.

[2]The abstract machine for PROTOS-L is an extension of the Warren Abstract Machine [Warren, 1983].

2 Requirements for the integration of transactions and database modifications into logic database programming languages

In this section we describe the requirements for the integration of transactions and database modifications into a logic programming language. The description focusses on how a given logic programming language can be extended to a logic database programming language. The requirements for the integration of transactions and backtracking will lead to corresponding requirements for the integration of database modifications and backtracking.

2.1 Requirements for the integration of transactions and backtracking

The integration of transactions into a logic database programming language has some requirements to both the embedding language and program execution. The embedding logic programming language shall have the following properties in order to integrate transactions:

1. The code of several transactions can be combined in one single logic program.
2. Transaction steps are programmed in the embedding logic programming language.
3. Integrity checking can be programmed in the embedding logic programming language.

The basic requirements to integrate transaction executions and backtracking are the following:

4. Transaction executions have to be atomic, i.e. atomicity of transaction executions has to be embedded in backtracking.
5. Database modifications of committed transaction executions have to be persistent, i.e. a concept of transaction persistence has to be integrated with backtracking.

Reqiurement 4 implies that all modification operations executed in a transaction have to be undone in case of a transaction abort. In order to meet this reqnirement, we will propose a semantics of write operations which is free of side effects in this case (c.f. sections 3.1 and 3.3).

Further, both atomicity and persistence of transaction executions require that backtracking is prevented from jumping inside an already committed transaction (c.f. section 3.2).

2.2 The integration of write operations and backtracking

The requirements for the integration of write operations and backtracking can be derived from the requirements 4 and 5 for the integration of transactions and backtracking:

6. Write operations are undone if the transaction they occur in is aborted, i.e. we want to avoid that write operations have side-effects that survive the transaction abort.

7. Write operations are made permanent if the transaction they occur in is committed.

Finally, we require what follows in order to keep programming in the language simple:

8. Write operations do not influence the flow of program execution, i.e. write operations are always successful (c.f. section 3.1).

3 A solution based on PROTOS-L

PROTOS-L [Beierle, 1989], [Beierle and Böttcher, 1989] is a typed logic programming language which is developed and implemented at IBM Stuttgart. PROTOS-L embeds read access to external databases, a module concept similar to that of Modula-2, and a polymorphic order-sorted type system supporting fast program execution. A detailed description of PROTOS-L as a database query language is given in [Böttcher, 1990a], whereas the PROTOS-L run time system is described e.g. in [Beierle, 1989], [Böttcher and Beierle, 1989] and [Böttcher, 1990b].

In this paper we concentrate on a proposal for the integration of database modification operations and a transaction concept into the programming language PROTOS-L. Subsection 3.1 and 3.3 describe how database modifications can be embedded, while subsections 3.2 and 3.4 outline how the suggested extension of PROTOS-L integrates transactions, in order to meet the requirements summarized in the last section.

3.1 A proposal for database modifications in the programming language PROTOS-L

Under the aspect of database access, PROTOS-L is basically intended to be a database query language, not to be a database modification langauge. Nevertheless, insert (:+) and delete (:\) operations can be embedded in the programming language PROTOS-L as follows. Insert and delete operations are goals, i.e. they can occur in any right hand side of a rule.

In order to describe the semantics of insert and delete operations we outline in this section,

- what are the preconditions for the operations,
- whether these operations sometimes fail or they have always success, and
- how the database state is changed when program execution passes these operations from left to right.

In section 3.3 we complete this description by explaining how the database state is changed when backtracking passes these operations from right to left.

We use the following example instead of giving a full set of definitions.

Let R be a database relation and $A_1, \ldots, A_n$ be arguments, i.e. A_i is either a variable or a constant. The precondition for the execution of the operations

R :+ ($A_1, \ldots, A_n$)   and   R :\ ($A_1, \ldots, A_n$)

is that each argument which is a variable is bound to a constant at the calling time of the operation. Hence, at calling time all arguments are bound to constants, say $c_1, \ldots, c_n$.

We further have to decide whether a delete operation (and an insert operation respectively) is always successful, or it is only successful if the tuple to be deleted (inserted) exists (does not exist) in the database relation. In order to meet requirement 8, insert and delete statements are always successful in the suggested extension of PROTOS-L, independent of the tuples currently stored in the database relation R.

The database state after the execution of the write operation (by passing it from left to right) is the same as the database state before the execution of the write operation with the following exception: If R_{pre} is the value of the relation R before the execution of a write operation, then

$R_{pre} \cup \{(c_1, \ldots, c_n)\}$

is the value of the relation R after the execution of the insert operation and

$R_{pre} \setminus \{(c_1, \ldots, c_n)\}$

is the value of the relation R after the execution of the delete operation.

The description how the database state is changed when backtracking passes database modifications will be delayed, until we have described the PROTOS-L transaction concept.

Finally, we give an example for a delete and an insert operation from a program part of our chemical production planning application. An execution of the following database rule changes the location, where a (chemical) product P is planned to be stored within the time interval [T1 , T2], from location1 to location2.

```
change_location ( P , T1 , T2 , 'location1' , 'location2' ) :-
        product_is_at :\ ( P , T1 , T2 , 'location1' ) ,
        product_is_at :+ ( P , T1 , T2 , 'location2' ) .
```

3.2 A proposal for transactions

In order to describe the scope of transactions, we suggest that the programming language PROTOS-L offers transaction brackets ({ and }) as language constructs, i.e. the action sequence of the transaction is enclosed in these transaction brackets. Note however, that the meaning of the transaction brackets ({ and }) differs from the meaning of the begin_transaction and end_transaction statements within procedural database programming languages, e.g. DBPL [Böttcher, 1989]. Further, the transaction brackets ({ and }) influence the flow of program execution and backtracking in order to meet requirements 4 and 5. We give the following example from our chemical production planning application[3] in order to discuss both aspects of the suggested transaction concept.

The transaction change_if_legal changes the planned storage location of a chemical product for the time intervall [T1,T2] from location L1 to location L2, only if the succeeding integrity_checks are successful.

```
change_if_legal(P,T1,T2,L1,L2) :-
    { , change_location(P,T1,T2,L1,L2) ,
      integrity_checks(P,T1,T2,L1,L2) , } .
```

After changing the location the integrity checks are performed. If they are successful, then the transaction is committed as soon as the } transaction bracket is passed from left to right. [4] On the other hand, if the integrity checks fail, then the transaction execution can not be completed successfully and the transaction is aborted when backtracking passes the left transaction bracket ({).

The integration of transaction brackets ({ and }) into backtracking described above can be summarized and generalized as follows:

- If program execution proceeds from left to right over a left transaction bracket ({), then the begin_of_transaction statement (tBegin for short) is performed.

- If program execution passes a left transaction bracket ({) from right to left (i.e. by backtracking), then the abort_of_transaction statement (tAbort for short) is performed.

[3] The requirements of this production planning application are summarised in [Böttcher, 1990c].
[4] At commit time the modifications of the transaction are made permanent to the database.

- If program execution proceeds from left to right over a right transaction bracket (}), then the commit_transaction statement (tCommit for short) is performed.

But, if program execution passes a right transaction bracket (}) from right to left (i.e. by backtracking), then the following problem arises: Backtracking from right to left over a right transaction bracket (}) has to be prevented from jumping to a choice point inside the transaction, because this transaction has already been committed.

The suggested solution to this problem is that, instead of jumping to a choice point inside a transaction backtracking has to jump to the last choice point allocated before the beginning of the transaction. Hence, the suggested solution meets requirement 4 for the integration of backtracking and transactions that transaction execution has to be atomic.

Like in other database programming languages, nested transaction calls are not allowed in this proposal. Since the modifications of transactions are made permanent to the database at commit time, and backtracking is prevented from jumping to a choice point inside a transaction, the proposed solution also meets requirement 5, that modifications of committed transactions are persistent.

3.3 The integration of database modifications and backtracking

In this setion we propose what has to be done, if backtracking goes from right to left over a database modification operation.

Requirement 6 states that all modifications executed in a transaction shall be undone, if the transaction is aborted, i.e. at the latest when backtracking reaches the left transaction bracket ({). The goal of this requirement is to avoid that write operations survive the transaction abort as side-effect of program execution.

The suggested solution to meet this requirement is as follows: Whenever backtracking passes a modification operation from right to left, then the modification operation is undone, i.e. backtracking reestablishes the database state given before this modification operation was executed. Hence, all database modifications are undone at transaction abort time, i.e. when backtracking reaches the left transaction bracket ({).

Note that insert and delete operations are different from assert and retract in Prolog [Cloksin and Mellish, 1981], because these modification operations are undone, as soon as backtracking passes these operations from right to left.

One the other hand, requirement 7 states that modifications of committed transactions have to be persistent. This can be achieved as follows: When program execution performs backtracking over a right transaction bracket (}), i.e. program execution goes to the last choice point activated before the transaction execution has begun, then the modifications made during transaction execution remain in the database.

To summarize: The only side-effect of write operations to the database during logic program execution is that side-effect which was required: modifications done in committed transactions are persistent.

3.4 A proposal for the implementation of transactions in the PROTOS-L compiler

The transaction concept can be implemented within the PROTOS-L compiler as follows. The compiler first translates PROTOS-L source code into some intermediate language, then it performs transformations and optimizations on this intermediate language representation of the program, and finally it translates the intermediate language representation into code for the PROTOS abstract machine.

At the intermediate language level, the code for transactions is transformed into logic programming code. The implementation of a transaction { ... } includes a cut in order to prevent backtracking from jumping from the outside of some transaction to a choice point allocated inside the transaction.

A transaction rule

```
transaction(P,T1,T2,L1,L2) :-
    { , transaction_body(P,T1,T2,L1,L2) , } .
```

can be implemented using the three built-ins tBegin, tCommit, and tAbort for the begin, the commit, and the abort of a transaction. These builts-ins are implemented by calls to the corresponding database system procedures of the underlying database system (which is in our case the SQL/RT database system). At the intermediate language level, the transaction rule is transformed into the following code:

```
transaction(P,T1,T2,L1,L2) :-
    transaction_implementation(P,T1,T2,L1,L2) .

transaction_implementation(P,T1,T2,L1,L2) :-
    tBegin , transaction_body(P,T1,T2,L1,L2) , tCommit , ! .

transaction_implementation(P,T1,T2,L1,L2) :- tAbort , fail .
```

Note that the cut behind the tCommit call prevents backtracking from jumping from outside the transaction to a choice point allocated by the transaction_body.

4 Summary and conclusion

The typed logic programming language PROTOS-L includes subtypes, polymorphism, a module concept and high-level read access to databases. We made a proposal how to integrate write operations and a transaction concept in order to extend this logic programming language to a database programming language. The transaction concept allows the application programmer to program transactions including the enforcement of integrity constraints within the logic programming language PROTOS-L. Several transactions can be combined in one single logic program. Further, the only side-effect of write operations during logic program execution is the side-effect which was required: modifications done in committed transactions are persistent. Hence, the suggested embedding of database modifications and transactions in the logic programming language PROTOS-L integrates backtracking and basic database programming concepts. The integration allows to implement integrated systems (using logic programming and embedding the knowledge of existing databases) in one single logic programming language. This opens up a large field of applications for logic programming languages.

References

[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. 2nd 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 and F. Gardin, editors, *Proc. International Symposium 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.

[Cloksin and Mellish, 1981] W.F. Cloksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, Heidelberg, New York, 1981.

[Fagin *et al.*, 1986] R. Fagin, G.M. Kuper, J.D. Ullman, and M.Y. Vardi. Updating logical databases. In P. Kannellakis, editor, *Advances in Computing Research*, Jai Press, 1986.

[Manchanda and Warren, 1988] S. Manchanda and D.S. Warren. A logic-based language for database updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, 1988.

[Schmidt, 1977] J.W. Schmidt. Some high level language constructs for data of type relation. *Transactions on Database Systems*, 2(3):247–261, 1977.

[Schmidt *et al.*, 1988] J.W. Schmidt, H. Eckhardt, and F. Matthes. *DBPL Report*. DBPL-Memo 111-88, Univ. Frankfurt, 1988.

[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.

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

[Wilkins, 1986] M.W. Wilkins. A model-theoretic approach to updating logical databases. In *Proceedings of the 5^{th} International Conference on Principles of Database Systems*, 1986.

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