Proceedings of the ISSEK-Workshop, Bled 1992

PLOP: A Polymorphic Logic Database Programming Language

Johann Eder
Universität Klagenfurt
Institut für Informatik
Universiätsstr. 65
A-9020 Klagenfurt / AUSTRIA

February 12, 1993

Extended Abstract

The language PLOP is a minimal language for studying polymorphism in a logical database programming language by means of parametrization of predicate names. The main idea is to introduce generic predicates which have two lists of parameters. While the second list contains the usual arguments, the first list contains parameters which are used as predicates in the rules defining the generic predicate. Through restriction in the use of predicate parameters it is possible to give a model theoretic semantics in a two step first order deduction. With the extensions we overcome the lack of predicate polymorphism in deductive database languages and increase their expressive power.

1 Introduction

Query languages for relational databases [Cod70, Cod72] are descendents of relational algebra or relational calculus, a subset of first order predicate logic [GMN84b]. They allow the formulation of query expressions in which the only names allowed are the names of database relations. To these languages we want to add well known powerful abstraction concepts, i.e. naming and parametrization. In the following we will concentrate on relational domain calculus, although this approach can also be applied to other languages [ERMS91].

Through the notion of views, naming had been introduced in relational languages to some extents. Views can be seen as a name assigned to a query expression, and this name can be used in other queries. However, the use of view names is restricted to avoid the formulation of recursive queries. So the use of view names is more like the use of macros, or like the use of subroutines in the sense of procedural programming, which does not fit well in the concept of declarative languages - relational calculus in our case.

Languages for deductive databases, Datalog in particular [CGT89, CGL90], extend relational calculus by allowing full use of the concept of naming in a declarative style. In Datalog, names of views (derived predicates) can be used in all query expressions like the names of base relations. However, Datalog does not permit the concept of parametrization, and thus leading to some disturbing shortcomings, especially the lack of polymorphism [BI90]. To give an example: Consider three base relations *Trains, Flights, Streets* each with the same attributes *fromCity, toCity.* For formulating the views *TrainConnection, FlightConnection, StreetConnection* (i.e. the transitive closures of the respective relations), in Datalog we have to define three derived predicates with six rules. The rules only differ in the name of the base relation. However, because there is no concept for using predicates as parameters, we can only use the text editor to take advantage of the conceptual identity of the rules. Of course, this problem is even worse in cases where the rules are more complex than our simple trabsitive closure example.

To overcome this problem we introduce the concept of predicate parameters in logic database languages. To avoid some problems of higher order logic we use a many sorted logic approach, so predicate names cannot be mixed with other arguments. Furthermore, we restrict the use of predicate parameters with some security conditions in a way, that we can define the semantics in a two step first order deduction.

The syntax is based on Datalog syntax in Prolog style notation. It is extended with the definition of generic predicates, which have two lists of parameters, the first list contains predicate parameters, the second the usual arguments.

The following example gives a first impression of such programs.

Examples of Plop-Programs:

```
% tc defines the transitive closure of a binary relation

tc(R/2)(X, Y) :-  R/2(X, Y).
tc(R/2)(X, Y) :-  tc(R/3)(X, Z), R/2(Z, Y).

% is there a  flight connection between vie and klu?

?- tc(direct-flights)(vie, klu).
```

```
% who are the ancestors of john?

?- tc(parent)(john, X).
```

The introduction of predicate parameters does not only add predicate polymorphism to logic database languages but it also increases their expressiveness. With the extension applied to stratified Datalog it is possible to formulate np-complete problems. With the extensions of an infinite integer type with the successor and predecessor function, the language is Turing complete.

In the following sections we first define the syntax of pure Plop and it's model theoretic semantics. We the discuss evaluation procedures for Plop and sketch some extensions.

2 Syntax

2.1 Names

We first define the following infinite alphabets

- *Const* a set of constants
- *Var* a set of variables ranging over Const
- *Pred* for predicate-constants
- *G-Pred* a set of generic predicate names
- *P-Var* a set of Variables ranging over predicates

Constants are denoted by strings starting with a lowercase letter, variables by strings starting with an uppercase letter. From the context it is always clear, whether a lowercase string is a constant, a predicate constant, or a generic predicate name, and whether a string with an upercase first letter is a variable or a predicate variable. For convenience we use names for predicate variables ending in /n, where n is the arity of this predicate variable.

With each predicate constant, each g-pred, and each predicate variable we associate a positive integer, the arity of the predicate. With each g-pred we associate another integer the p-arity.

2.2 Atoms, Literals, etc.

As usual we define a *term* to be either a variable or a constant, and $p(t_1, \ldots t_n)$ as *literal*, where p is an n-ary predicate symbol and $t_1, \ldots t_n$ are terms.

Facts are literals. *Rules* are represented in the general pattern:

$$L_0 : -L_1, \ldots, L_n$$

Each L_i is a literal. We call L_0 the head of the rule and $L_1, \ldots, L_n$ the body of the rule. A literal, fact, rule, or clause without any variable is called ground.

If $p_1, \ldots, p_n$ are predicate constants or predicate variables and c is a g-pred of p-arity n, then $c(p_1, \ldots, p_n)$ is an *elementary p-term.* If $p_1, \ldots, p_n$ are predicate constants or predicate variables, or p-terms, and c is a g-pred of p-arity n, then $c(p_1, \ldots, p_n)$ is a *p-term.* c is called the *main predicate* of the p-term. The *arity of a p-term* is the arity of it's main predicate.

If s is an elementary p-term with arity n and $t_1, \ldots, t_n$ are terms, then $s(t_1, \ldots, t_n)$ is an elementary g-literal.

We define a g-literal (generic literal) as being either a literal or an expression of the general pattern $c(t_1, \ldots, t_n)$, where c is a predicate, a predicate variable, or a p-term of arity n, and $t_1, \ldots, t_n$ are terms.

If a g-literal contains no variables, we call it *ground*, if it contains no predicate variables we call it *instantiated.*

G-rules are represented in the pattern

$$G_0 : -G_1, \ldots, G_n$$

Each $G_i, i \geq 1$, is a g-literal, G_0 is literal or an elementary g-literal.

A *goal* is an instantiated g-literal.

A *Plop program* consists of a set of facts and a set of g-rules, and a goal. The set of facts is called the extensional database (EDB) and the set of g-rules the intensional database (IDB). No predicate of a fact is allowed to be predicate of a head literal of a rule of the IDB.

For writing PLOP programs we use Prolog notation.

In the following we require the following safety conditions:

1. All facts are ground.
2. All variables occuring in the head of a rule must also appear in the body of a rule.
3. All predicate variables appearing in the body of a rule also appear in it's head.

2.3 Rule Instances

A g-rule with all predicate variables instantiated with predicate names or ground p-terms are called rule instances.

Due to third safety condition the substitution of predicate variables in the head of a rule suffices to specify the rule instances. So given an instantiated literal l with a g-pred p as main predicate, we derive the set of instantiated rules for l by by substituting the predicate parameters in all literals in all rules with p as main predicate of the head literal with the predicate arguments of l.

2.4 Extensions of pure Plop

We define extensions of pure Plop: evaluable predicates, and stratified negation the same way they are defined in Datalog. However, in this paper we will concentrate on pure Plop.

3 Model theoretic semantics

The Plop-Base is defined as the set of all predicates and all instantiated p-terms.

A Plop-Interpretation is a subset of the Plop Base.

A Plop-Interpretation is a Model for a Plop-program, iff it satisfies all g-rules. It satisfies a g-rule, iff it satisfies all instances of that g-rule, and it satisfies an instance of a g-rule, iff it holds, that if all (instantiated) literals in the body of the rule are in PM, so is the (instantiated) head-literal.

Theorem: The intersection of two Plop Models is a Plop Model.

The least Plop model is the intersection of all Plop models.

The least Plop Model is a set of predicate names and instantiated p-literals. The intended program of a Plop program is the set of all rules and the set of all instantiated rules induced by the instantiated p-terms in the least Plop model. The intended program of a Plop program does not contain any predicate variables. If we consider all instantiated p-terms as strings denoting a predicate name, the intended program is a Datalog program (possibly an infinite one due to recursive nesting of p-terms).

The semantics of a Plop program is now defined as the semantics of the intended program interpreted as Datalog program.

4 Evaluation

For evaluating Plop programs we have to distinguish different cases:

1. *finite intended program*

   If the intended program is finite, we can deduce the intended program in a first step and evaluate the inted program in a second step. We check whether the intended program is finite by dedecting cycles in an extended dependency graph.

   The definition of generic transitive closure is an example for a finite intended program.

2. *partitionable program*

   In this class the number of rule instances is infinite. However, it is possible to partion the infinite set of instantiated generic predicates into equivalence classes according to their *extensions*, i.e. two instantiated generic predicates with identical main predicate belong to the same equivalence class, if they are true for the same substitutions, i.e. if their extensions are identical. We can then substitute instantiated generic predicates of the intended program with the the member of the same equivalence class with the least nesting depth. If the set of equivalence classes is finite, the intended program is replaced with an equivalent finite program.

   Example of a partionable program:

   ```
   g(R/2)(X,Y) :- g(p(R/2))(X,Z), p(R/2)(Z,Y).
   p(R/2)(X,Y) :- R/2(X,Y), q(Y).

   ?- g(r)(X,Y).
   ```

   It is easy to see that the intended program is infinite, and $\{g(p^i(r)), p^i(r)\}$ is it's set of instantiated generic predicates. We can substitute $p^i(r)$ with $p^j(r)$, where j is the smallest number such that $p^i(r) = p^j(r)$, where $=$ means extensional equality. As the predicates p and q are base relations, and all constants in the possible ground literals of $p^i(r)$ are also in p and q, $\{p^i(r)\}$ can be partitioned into a finite set of equivalence classes, and, therefore, the program is finitely evaluable.

3. *finite evaluable program*

This class is especially relevant for Plop programs with stratified negation. It uses the fact, that if any literal in the body of a rule is empty, the other literals need not be evaluated and thus the infinite intended program is replaced by an equivalent finite one. It has to be noted, that this equivalent program can only be determined during the evaluation and not beforehand.

4. *infinite program*

   A fourth class remains, containing all programs which cannot be evaluated finitely.

5 Conclusion

In this paper a logic database language is introduced which extends relational calculus with the abstraction concepts naming and parametrization. The language is based on Datalog, it's extensional, however can also be integrated in other query languages. The extensions overcome the lack of predicate polymorphism of query languages. Therefore, the main application of the extensions is supposed be the subclass of programs which yield a finite intended program.

Currently a prototype implementation of the language is under way, where the optimized QSQ algorithm for the evaluation of Datalog programs [Vie86, Vie88] is extended for evaluating Plop programs. Furthermore, efficient algorithms are developed to determine, whether a Plop program can be evaluated finitely.

References

[BI90] A.J. Bonner, and T. Imielinski. The reuse and Modification of Rulebases by predicate substitution. In F. Bancilhon, et al., editors, *Advances in Database Technology - EDBT'90 (Proceedings)*, Springer-Verlag, 1990.

[CGL90] S. Ceri, G.Gottlob, and L.Tanca. *Logic Programming and Databases.* Springer Verlag, New York, 1990.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. on Knowledge and Data Eng.*, 1(1), 1989.

[Cod70] E. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6), 1970.

[Cod72] E. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[ERMS91] J. Eder, A. Rudloff, F. Matthes, and J.W. Schmidt. Data construction with recursive set expressions. In J. W. Schmidt and A.A. Stogny, editors, *Next Generation Information System Technology*, LNCS 504. Springer Verlag, 1991.

[GMN84b] H. Gallaire, J. Minker, and J-M. Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 2, 1984.

[Vie86] L. Vieille. Recursive axioms in deductive databases: The Query-Subquery approach. In L. Kerschberg, editor, *Proc. Int. Conf. Expert Database Systems*, Charlston, 1986.

[Vie88] L. Vieille. From QSQ to QoSaQ: Global optimization of recursive queries. In *Proc. 2nd Int. Conf Expert Database Systems*, Tyson Corner, 1988.