IMPLEMENTATION OF RECURSIVE QUERIES FOR A DATA LANGUAGE BASED ON PURE HORN LOGIC

Domenico Saccà †
University of Calabria, Rende, Italy
Carlo Zaniolo
MCC, Austin, Texas, USA

Abstract

This paper treats the problem of implementing efficiently queries expressed by Horn clauses containing recursive predicates, including those with function symbols. In particular, the situation is studied where the initial bindings of the arguments in the recursive query goal can be used in the top-down (as in backward chaining) execution phase to improve the efficiency and, often, to guarantee the termination of the forward chaining execution phase that implements the fixpoint computation for a recursive query. (To ensure efficient support for database applications this fixpoint computation is actually carried out by relational algebra operators.) A general approach is given for solving these queries; the approach performs an analysis of the binding passing behavior of the query, and then reschedules the overall execution as two fixpoint computations derived as results of this analysis. One such computation emulates the propagation of bindings in the top-down phase; the other generates the desired answer by proving the goals left unsolved, in the recursive rules, by the previous step. Finally, sufficient conditions for safety are derived, to ensure that the fixpoint computations are completed in a finite amount of time. Two instances of this basic approach are discussed in details: one is called the magic set method and the other is the minimagic method.

---

† Part of this work was done while this author was visiting at MCC.

1. Introduction

While the advantages of Logic in studying and formally describing knowledge-based computing have long been recognized [Nil, GMN], its use in programming actual applications was hampered, for a long time, by the lack of efficient implementations. This situation changed with the introduction of Prolog which proved the amenability of Horn Clause Logic to efficient implementation, and the applicability of Logic Programming to a variety of realms, including expert systems, translator writing, and databases.

For all its merits, however, Prolog must be regarded as a still imperfected realization of the Logic Programming idea, since it is based on a particular execution model -- SLD resolution with left-most goal expansion-- which is not logically complete. Furthermore, non-logical constructs (such as the cut, assert and retract) where included in the language using the sequential execution model as the basis for their meaning. Therefore, Prolog is, to a large extent, a prescriptive language which suffers from the following drawbacks:

(a) The programmer must guarantee the performance and termination of the application at hand by carefully ordering rules and goals; this detracts from the ease of use of the language, and from the portability and generality of applications written in it.

(b) The Prolog execution model is well-suited for main memory resident fact bases and single processor machines, but tends to be inefficient for fact bases stored in secondary memory (e.g., for database applications); moreover, the imperative constructs in Prolog limit its amenability to parallel processing.

A strategy for overcoming the limitations of point (b) has been that of extending the language with additional constructs designed for interfacing to databases [KY, Z1]. or for controlling parallel execution [CG,ST]. While such an approach is undeniably one of great expedience and short-term benefits, it also represents a further move away from the elegance of declarative logic, and leads to an endless stream of ad-hoc solutions and potpourri semantics. Moreover, this strategy offers no relief with respect to point (a).

In this paper, instead, we suggest to return to the pristine -- minimum model and fixpoint based -- semantics of definite Horn clause queries [VK, Llo] and to have the system select a safe and

efficient execution strategy for the given query and processing environment.[1] A number of interesting proposals have been advanced for solving the formidable technical problems thus issuing, including the use of meta-logic [SG], and of dynamic goal reordering [Nai]. More directly germane to this paper is what can be termed a "database oriented" approach; a number of recent contributions can, tentatively, be cataloged in this class [Ban, BMSU1, GD, KL, U2, SZ1,Vi]. In addition to the Logic Programming experience, the database oriented approach has been influenced by the work on deductive databases [Rei, KT], whereby the intensional information (query and rules) is treated separately from the extensional one (fact base); the intensional information is compiled into efficient code that is applied at run time to the extensional database. The third important influence comes from database systems, that during the last decade, have evolved from navigational DB systems (based on a sequential execution model very similar to that of Prolog [Z1]) to relational systems which use declarative query languages based on logic. This recent progress of database systems suggests that a similar evolution may take logic based languages past Prolog and towards truly non-procedural languages. Moreover the relational database experience can provide many techniques that will be useful in solving the arduous technical problems thus resulting [Par, U2].

The "database oriented" viewpoint has greatly influenced the design and the implementation of LDL which is a language based on Horn clause logic and intended for data intensive and knowledge based applications [TZ]. Thanks to the amalgamation of the functionalities of logic programming and databases, an LDL system will be able to support and manage large data and rule sets and remove the "impedance mismatch" between the query language and the programming language, currently besetting the development of database applications [CM].

---

[1] This approach remains applicable when useful declarative extensions, such as sets and negation, are included [TZ]. However, inasmuch as these extensions are based upon stratification assumptions [BNRST], they are orthogonal to the recursion issues discussed here.

2. Technical Issues

Consider a grandmother rule;

$GM(x,y)$:- $P(x,z)$, $M(z,y)$.

with parent defined by a father and mother base relations [2]:

$P(x,y)$:- $F(x,y)$.
$P(x,y)$:- $M(x,y)$.

Then, an obvious way to support a query

G1: -?$GM(x,y)$.

is to use bottom up processing; using relational algebra [U1] this can be expressed as follows (take the union of F and M, then join the result with M and project out the middle column):

$$GM = \pi_{1,4}((F \bigcup M) \rhd\lhd_{2=1} M)$$

where F and M denote the relations that store the father and mother fact base. The important fact to be observed here is that by using a bottom-up processing we have replaced full unification with matching (only one of the two terms contain variables) which is a computationally more efficient and parallelizable operation [DKM, MK]. Since we are interested in data intensive applications, we will implement matching using relational algebra operators, since these have proven effective in database applications, but this is only incidental to the discussion of this paper. However, let us consider a second query

G2: -?$GM(marc, y)$.

We could support this second query by filtering the results of the first by a selection condition: $\sigma_{1=marc}$; however, this approach is inefficient, since only the mother and father of marc need to be considered in taking the union and the successive join; i.e., the selection should be pushed down and applied directly to the fact base. The Prolog solution to this problem is a two-phase process whereby the constant is first migrated down during the depth-first goal expansion, and a second phase where the actual join is performed

[2] A predicate that only unifies with facts will be called a database predicate. By base relation or database relation we mean a set of facts having the same predicate symbol and number of arguments.

and the answer returned. The price paid by Prolog is the need to use full unification in the first phase. Thus, we opt for the alternate solution of pushing the constant down at compile time by modifying the original rules for the particular query. Thus our query is reformulated into

G3: -?GM(marc, y).
GM(marc,y) :- P(marc,z), M(z,y).
P(marc,y) :- F(marc,y).
P(marc,y) :- M(marc,y).

As we turn this specialized set of rules into its relational algebra equivalent we see that the selection is now applied directly into the database. Thus, the query compilation approach just described offers the important advantages of (i) eliminating Prolog's first computation phase, and (ii) requiring matching rather than full unification.

Unfortunately, things become more complicate when recursive predicates are involved. Take for instance the recursive predicate *SG* of Figure 1,

r_0: $SG(x,y) :- P(x,x_1), SG(x_1,y_1), P(y,y_1)$
r_1: $SG(x,x) :- H(x)$.

Fig. 1. *The same-generation example.*

where $P(x,x_1)$ is a database predicate describing that x_1 is the parent of x, and $H(x)$ is a database predicate describing all humans. We want to support recursive queries by a least fixpoint operator; this decision is in natural agreement with (a) the semantics of Horn clauses that is defined using the notions of minimum model and least fixpoint [VK, Llo], (b) the bottom up, matching-based execution strategy and (c) the operator-based approach, using relational algebra.

For instance, for the query,

G4: ?- SG(x,y)?

one will start by setting the initial value of a variable relation SG to empty and computing a new value, say

$$SG' = SG \bigcup f(SG)$$

where f denotes the bottom up computation defined by the previous

rules. Using relational algebra, for instance, f can be expressed as follows:

$$f\ (SG) = \pi_{1,1} H \bigcup \pi_{1,5}((P \rhd\lhd_{2=1} SG) \rhd\lhd_{4=2} P))$$

Thus, after replacing SG by SG', the computation iterates and terminates when $SG' = SG$. Since our rules contain only positive goals, the function f is monotonic and continuous in the lattice of relations defined by set containment, and this procedure computes a unique least fixpoint [Tar]. This fixpoint approach, refined with the differential techniques described in [BaR, Ban], is effective for queries as G4, where no argument is bound. However, the basic fixpoint approach becomes inefficient once the recursive predicate is called with some arguments bound. For a query,

$G5$: $?-SG\,(adam, y)$

for instance, the fixpoint approach will still compute all possible pairs of humans that are of the same generation, only to later discard those that do not have "adam" has their first argument. Clearly, we need a strategy for taking advantage of the constants present in the query. Unfortunately, the simple approach of specializing the rules by substituting the constants in place of the variables does not work for recursive predicates. For instance, if we replace the occurrence of the variable x by "adam" we obtain rules that do not produce humans of the same generation as adam. (Of course, there are also cases in which the substitution trick works [AU]; but detecting those cases is, in general, undecidable [BKBR].) Thus, we need some new techniques for taking advantage of constants in recursive queries. The need for such techniques is reinforced by the safety issues discussed next.

To illustrate a first aspect of the safety issue let us assume that the goal (i.e., $H(x)$) is removed from rule r_1 in Figure 1. Then, assuming an infinite underlying universe, the query $G4$ becomes unsafe, since any pair (x, x) satisfies it. The query $G5$ is safe per se, since only a finite number of people can be of the same generation of adam; but its evaluation using the fixpoint approach is still unsafe, and therefore unfeasible. Here, an evaluation that is capable of migrating the query constraints downwards is needed to ensure the very feasibility, rather than efficiency, of the execution strategy.

The safety issue is even more of a concern for "computational" predicates such as that of Figure 2, which are normally intended for use only with certain input bindings. The potential sources of unsafe behavior in the rules of Figure 2 are two. One is that the non-recursive rules r_2 and r_3 admit infinite models --this is the same problem as in the previous example. However, even if the x-values range over a finite set, the recursive rules r1 and r2 generate longer and longer lists at each step in the fixpoint computation, which, therefore, never ends.

r_0: $MG(x \bullet y, x_1 \bullet y_1, x \bullet w) :- MG(y, x_1 \bullet y_1, w),\ x \geq x_1$
r_1: $MG(x \bullet y, x_1 \bullet y_1, x_1 \bullet w) :- MG(x \bullet y, y_1, w),\ x < x_1$
r_2: $MG(nil, x, x)$
r_3: $MG(x, nil, x)$

Fig. 2. *Merging two sorted lists.*

In conclusion, an effective usage of the bindings information available in a recursive goal is vital for performance reasons and to avoid the non-termination pitfall. Therefore, it should be of no surprise that a number of interesting approaches were proposed to deal with this problem, including [BMSU1, GD, CH, HN, KL, MS, SZ1, U2, Vi]. The reader is referred to [BR] for an overview and a comparison of these approaches. As described in [BR], some of these approaches lack generality --i.e., the realm in which they work is either narrow or poorly defined-- and they lack robustness -- i.e., they are stated in terms of specialized algorithms or lower level primitives, hence their suitability to different execution models is unclear.

In this paper, we propose a formal framework and general algorithms to guarantee safety and efficiency in the implementation of recursive predicates. The proposed solution takes the form of rewriting rules whereby the original query and relative rules are recast into an equivalent set that can be safely and efficiently implemented using least fixpoint operators and matching. Since these are generic high level operations, an assortment of more specific architectures and implementation primitives can then be used at a lower level; in particular, relational algebra operators are suitable for implementations intended for data intensive application environments.

Two methods are presented for the compilation of recursive predicates: one is the *magic set method* [BMSU1, BMSU2, ZS1] that is here extended to handle function symbols, comparisons and nested recursive predicates. The second is a new method called *minimagic method* that improves on the previous one by eliminating duplicate work. The two methods can be developed along a common framework consisting of following steps:

i) a symbolic analysis of the binding propagation behavior during the top-down phase, and using the results of this analysis,

ii) the generation of rules for the computation of special sets (i.e., the magic sets or the minimagic sets) that actually implement the top-down propagation of bound values,

iii) the generation of rules for producing the query answer.

The two methods share i) and they differ in the specific rules generated in (ii) and (iii). The paper is organized as follows. We first describe point i) and then we present the two methods. Finally we discuss their safety and their trade-offs.

3. Binding Passing Property.

In a logic program LP, a predicate P is said to *imply* a predicate Q, written $P \rightarrow Q$, if there is a rule in LP with predicate Q as the head and predicate P in the body, or the exists a P' where $P \rightarrow P'$ and $P' \rightarrow Q$ (transitivity). Then any predicate P, such that $P \rightarrow P$ will be called *recursive*. Two predicates P, and Q are called *mutually recursive* if $P \rightarrow Q$ and $Q \rightarrow P$. Then the sets of all predicates in LP can be divided into recursive predicates and non-recursive ones (such as database predicates). The implication relationship can then be used to partition the recursive predicates into disjoint subclasses of mutually recursive predicates, which we will call *recursive cliques,* with their graph representation in mind. All predicates in the same recursive clique must be solved together — cannot be solved one at a time.

For the LP of Figure 1, SG is the recursive predicate (a singleton recursive clique), and H and P are database predicates. However, in the discussion which follows, H and P could be any predicate that can be solved independently of SG; thus they could be derived predicates — even recursive ones— as long that they are not

mutually recursive with SG. Finally, it should be clear that "john" is here used as a placeholder for any constant; thus the method here proposed can be used to support any goal with the same binding pattern.

Formally, therefore, we will study the problem of implementing a query Q that can be modeled as triplet $<G, LP, D>$, where:

LP is a set of Horn clauses, with head predicates all belonging to one recursive clique, say, C.

G is the goal, consisting of a predicate in C with some bound arguments.

D denotes the remaining predicates, in the bodies of the LP-rules, which are either non-recursive or belong to recursive cliques other than C.

The predicates in C will be called the *constructed predicates* (c-predicates for short) and those in D the *datum predicates*. For instance, if our goal is $G\,2{:}SG(john, x)?$ on the LP of Figure 1, then SG is our c-predicate (a singleton recursive clique) and P and H are our datum predicates.

In general, datum predicates are those that can be solved independently of the c-predicates; therefore, besides database predicates they could also include predicates derived from these, including recursive predicates not in the same recursive clique as the head predicates. Take for instance the LP of Figure 2, with goal

$$MG(L_1, L_2, y)?$$

where L_1 and L_2 denote arbitrary given lists. Here MG is our c-predicate and the comparison predicates $\geq$ and $<$ are our datums. The $<$ predicate could, for instance, stand for a database predicate (e.g., if there is a finite set of characters and their lexicographical order is explicitly stored: $a < b$, $b < c$, $\cdots$) or it could stand for a built-in predicate that evaluates to false or true when invoked as a goal with both arguments bound, or, with integers defined using Peano's axioms, it could be the recursive predicate of Figure 3,

r_0: $x < s(x)$.
r_1: $x < s(y) :- x < y$.
Fig. 3. *The "less-than" relationship for integers represented using the successor notation.*

Exit rules and recursive rules:

A rule with a recursive predicate R as its head will be called *recursive* if its body contains some predicate from the same recursive clique as R; it will be called an *exit rule,* otherwise.

For notational convenience, we will always index the recursive rules starting from zero, $r_0, \cdots, r_{m-1}$; thus, the total number of recursive rules under consideration is always m. For instance, in Figure 2, r_0 and r_1 are the recursive rules, while r_2 and r_3 are the exit rules.

3.1. Binding Propagation

Datum predicates propagate bindings from the bound arguments in the heads of the rules to arguments of the c-predicate occurrences in their bodies. Let us, *for now,* say that our only datums are database and comparison predicates; then the binding propagation in a rule r_i can be defined as follows. Say that B is a set of (bound) variables of r_i. Then *the set of variables bound in* r_i *by* B will be denoted B_{r_i+} (or B^+ when r_i is understood) and is recursively defined as follows:

i) (basis)

Every variable appearing in B is also in B^+

ii) (induction)

database predicates: If some variable in database predicate is in B^+ then all the other variables are in B^+.

comparison predicates: If we have an equality, such as $x = expression$ or $expression = x$, and all the variables in *expressions* are in B^+, then x is in B^+ as well.

Let P be a predicate in the body of r_i. Then, an *argument* of P will be said to be *bound* by B when all its variables are bound by B. If all arguments of P are bound by B then the predicate P is solved by B.

Consider again a rule r_i. Let S denote a set of bound arguments in the head of r_i. Then we say that r_i is *solved by* S if all its variables are bound by the variables in $B_S \cup B_c \cup B_d$ where:

B_S are the bound variables in the head (i.e., those contained in the S-arguments),

B_c are the variables of c-predicates in the body of r_i (if any), and

B_d are the variables of database predicates in the body of r_i.

Let S be the bound arguments in the head predicate of r_i and B the (bound) variables in these arguments. Moreover, let T denote the set of arguments bound by B in a c-predicate occurrence P. Then we will say that r_i *maps the set of bound arguments* S *of its head, into the set of bound arguments* T *of* P.

Say for instance that the first argument of SG is bound in Figure 1. Then $B = \{x\}$, i.e., x is bound. Moreover, x_1 is bound by B via the database predicate P. Thus in r_0 of Figure 1, the bindings propagate from SG^1 to SG^1. Thus $P(x, x_1)$ is a solved predicate in r_0, whereas $P(y, y_1)$ is not. $H(x)$ is solved in r_1.

3.2. Binding Graph of a Query

The *binding graph* of a query is a directed (multi-)graph having nodes of the form P^S where P is a c-predicate symbol and S denotes its bound arguments, and whose arcs are labeled by the pair $[r_i, v]$, where r_i is the index to a recursive rule, and v is a zero-base index to c-predicate occurrences in the body of this rule, i.e, 0 is the index to the first c-predicate occurrence, 1 to the second one, etc. (the zero base is chosen to simplify the counting operations). The binding graph M_Q for a query $Q = < G, LP, D >$ is constructed as follows:

i) If S is the non-empty set of bound arguments in G, then G^S is the *source node* of M_Q,

ii) If there exists a node R^S in M_Q and there is a (recursive) rule r_i in LP that maps the bound arguments of R into the bound arguments T of the v-th c-predicate occurrence and this has symbol P, then P^T is also node of M_Q, and there is an arc labeled $[r_i, v]$ from R^S to P^T.

Figure 4 shows a binding graph for a query $SG^{1,2}$ on the rules of Figure 1, and Figure 5 shows the graph for a query $MG^{1,2}$ on the rules of Figure 2.

We can now enunciate our key property.

Binding passing property:
A query Q will be said to have the *binding passing property* when the following properties hold for each node R^S of its binding graph:

(a) S is not empty, and

(b) each (exit or recursive) rule r_i such that the predicate symbol of its head is R, is *solved* by S.

The binding passing property guarantees that (a) the bindings can be passed down to any level of recursion, and that (b) all predicates in the recursive rules can be solved either in the top-down or in the bottom-up execution phase. Our examples in Figures 1 and 2, with binding graphs of Figure 4 and 5, have the binding passing property. However, the query

R(a,y)?
R(x,y) :- R(w,x), B(y), y>w.
R(x,x) :- B(x).

does not have the binding passing property since $R^1 \rightarrow R^2 \rightarrow R^\phi$ Our binding graph is similar to the rule/goal graph described in [U2] and is an extension of the query binding graph presented in [SZ1].

We point out that we assume that the binding passing property needs to be checked only once for any given binding pattern in the query (e.g., at compile time), moreover the following proposition guarantees that binding graphs can be constructed efficiently.

PROPOSITION 1 [SZ2]. *Let $Q = <G, LP, D>$ be a query such that there is a bound on the arity of the predicates in LP, then*

a) *The binding graph of Q can be constructed in time linear in the size of LP and G.*

b) *The binding passing property of Q can be tested in time linear in the size of LP.* □

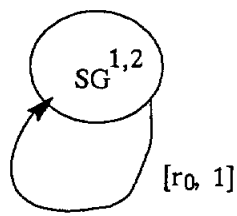
G : SG(a, b)?

LP:

r_0: SG(x, y):- P(x, x_1), P(y, y_1), SG(x_1, y_1).

r_1: SG(x, x):- H(x).

Binding Graph



Magic Rules

magic.SG$^{1\ 2}$(a, b).

magic.SG$^{1\ 2}$(x_1, y_1) :- magic.SG$^{1\ 2}$(x, y), P(x,x_1), P(y, y_1).

Modified Rules

SG$^{1\ 2}$(x, y) :- magic.SG$^{1\ 2}$(x, y), P(x, x_1), P(y, y_1), SG$^{1\ 2}$(x_1, y_1).

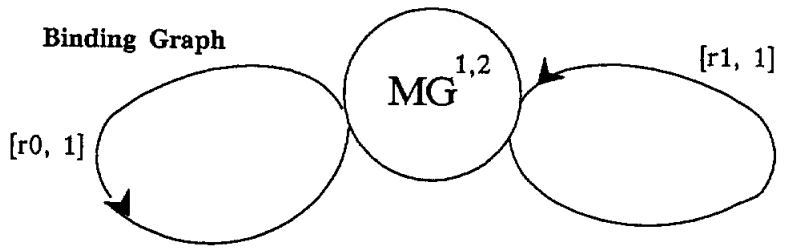
SG$^{1\ 2}$(x, x) :- magic.SG$^{1\ 2}$(x, x), H(x).

Figure 4. *The magic set method for SG(a,b)?*

G: MG(L_1, L_2, W)?

LP:

r0: MG(x•y, x_1•y_1, x•w) :- MG(y, x_1•y_1, w), $x \geq x_1$.
r1: MG(x•y, x_1•y_1, x_1•w) :- MG(x•y, y_1, w), $x < x_1$.
r2: MG(**nil**, x, x).
r3: MG(x, **nil**, x).



Magic Rules

magic.MG1 2(L_1, L_2).

magic.MG1 2(y, x_1•y_1) :- magic.MG1 2(x•y, x_1•y_1), $x \geq x_1$.

magic.MG1 2(x•y, y_1) :- magic.MG1 2(x•y, x_1•y_1), $x < x_1$.

Modified Rules

MG1 2(x•y, x_1•y_1, x•w) :-
magic.MG1 2(x•y, x_1•y_1), MG1 2(y, x_1•y_1, w), $x \geq x_1$.

MG1 2(x•y, x_1•y_1, x_1•w) :-
magic.MG1 2(x•y, x_1•y_1), MG1 2(x•y, y_1, w), $x < x_1$.

MG1 2(**nil**, x, x) :- magic.MG1 2(**nil**, x).

MG1 2(x, **nil**, x) :- magic.MG1 2(x, **nil**).

Figure 5. *Magic Set Implementation of the List Merge*

4. The Magic Set Method.

We now present a method to implement logic queries which have the binding passing property defined in the previous section. The method presented next extends the magic set method described in [BSMU1, BSMU2] and [SZ1] to include function symbols, comparison predicates, and nested recursive queries.

The objective of the general magic set method is to transform the query $Q = < G, LP, D >$ into a query $\overline{Q} = < \overline{G}, \overline{LP}, D >$ which has the same answer as Q. This transformation is performed at compile time, following, and guided by the results of, the binding graph analysis. A set of new rules, called the *magic rules*, defining various *magic predicates* is generated. Moreover, every rule of LP is transformed into a *modified rule* by the addition of a magic predicate to the rule body. At run time, the magic rules execute the propagation of bindings, while the successive bottom-up evaluation of the modified rules computes the answer, having some variables constrained to range on sets generated by the previous phase.

4.1. Magic Rules

A number of new predicate symbols are introduced, one for each node of the binding graph M_Q of Q. They correspond to sets of values (*magic sets*) which contain the propagated bindings. The magic set corresponding to a node R^S of M_Q has arity equal to $|S|$ and is denoted by $magic.R^S$. The magic sets are computed by means of the magic rules.

Exit Magic Rule:

The first magic rule is generated by the source node in M_Q, say P^S which corresponds to the query goal. Say that the query goal has $n = |S| \geq 1$ bound arguments with respective values $a_1, \cdots, a_n$. (Here and elsewhere, arguments as well as variables are listed in the order they appear in the predicate or in the rule.) Then we add the following clause for the magic set:

$$magic.P^S(a_1, \cdots, a_n).$$

Recursive Magic Rules:

There is a recursive magic rule for each arc in M_Q as follows: for an arc labeled $[r_i, j]$ from node R^S to node P^T, we add the rule

$$magic.P^T(y_1, \cdots, y_l) :- magic.R^S(x_1, \cdots, x_n), Q_1, \cdots, Q_h$$

where:

i) $x_1, \ldots, x_n$ are the bound arguments in the head of r_i (i.e., those in S),

ii) $y_1, \cdots, y_l$ are the bound arguments in the j-th c-predicate of r_i (i.e., those in T),

iii) $Q_1, \cdots, Q_h$ are the datum predicates of r_i, solved by the variables in the bound arguments S.

Informally described, the magic rules are constructed by removing all unsolved datum predicates, all c-predicates but the j-th one, by exchanging this c-predicate with that in head, and by removing the bound arguments in these two predicates. Note that, while there are as many magic rules as arcs in the graph, their head predicates are only as many as there are nodes in the graph — see Figures 5 and 6 for an example.

4.2. Modified Rules and Goal:

Modified Recursive Rules:

A number of new predicate symbols are introduced, one for each node in M_Q, to replace the c-predicate symbols in LP. For each node in M_Q, there are as many modified rules as there are bundles of arcs labeled with the same rule, leaving the node. Thus, let R^S be a node in M_Q and r_i be the label of a bundle of arcs leaving R^S (thus r_i is a recursive rule with head predicate symbol R); then, each original rule r_i is modified by the addition of a magic predicate to the body of the rule, as follows:

$$R^S(x_1, \cdots, x_n) :- magic.R^S(z_1, \cdots, z_l), P_1, \cdots, P_k, Q_1, \cdots, Q_h$$

where:

(i) $z_1, \cdots, z_l$ are the bound arguments in the head of r_i (i.e., those in S),

(ii) $P_1, \cdots, P_k$ are the original c-predicates each adorned by its bound arguments, and

(iii) $Q_1, \cdots, Q_h$ are the remaining predicates, i.e., all datum predicates.

Modified Exit Rules:

Say that R^S is a node of M_Q and there is an exit rule r_i with head predicate symbol R. Then we modify the original rule r_i by

(i) replacing the predicate symbol R by R^S, and

(ii) adding the goal

$$magic.R^S(x_1, \cdots, x_n)$$

to its body, where $x_1, \cdots, x_n$ are the bound arguments in the head of r_i

Modified Goal

If the source node in M_Q is P^S then the modified goal becomes $P^S(x1, \cdots, x_n)$ where $x_1, \cdots, x_n$ denote all arguments (bound or unbound) in the query goal.

Examples of the Magic Set method are in given in Figures 4, 5 and 6.

4.3. Properties of the Magic Set Method

From a formal viewpoint, the Magic Set Method can be viewed as a rule rewriting system. In this framework, both the original set of Horn Clauses and the modified one have a pure fixpoint-based semantics that defines the sets of answers satisfying the query [VK] (arithmetic predicates can be treated in this framework as being defined by infinite comparison relations over complex arithmetic terms [Z3]). Then we can prove the following basic result [SZ2]:

THEOREM 1. *Let* $Q = <G, LP, D>$ *be a query and* $\overline{Q} = <\overline{G}, \overline{LP}, D>$ *be the query modified by the magic set method. Then* Q *and* $\overline{Q}$ *compute the same answer.*

PROOF. The proof follows the lines of that given in [SZ2] for showing that the query modified by the generalized counting method computes the same answer as Q. □

Theorem 1 only establishes the logical correctness of $\overline{Q}$. The fact that $\overline{Q}$ has the desired characteristics, e.g., uses only matching, is guaranteed by the binding passing property. When this property holds, the magic set method prescribes an abstract computation

G : P(a, y)?

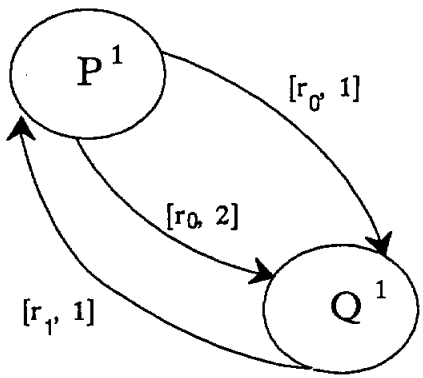
LP :

r_0: P(x, y) :- B1(x, w, x_1), Q(x_1, y), B2(w, x_2), Q(x_2, y), B3(y, z).

r_1: Q(x, y) :- B4(x, z), P(z, y).

r_2: P(x, y) :- B5(x, y).

Binding Graph



Magic Rules

magic.P1(a).

magic.Q1(x_1) :- B1(x, w, x_1), B2(w, x_2), magic.P1(x).

magic.Q1(x_2) :- B1(x, w, x_1), B2(w, x_2), magic.P1(x).

magic.P1(z) :- B4(x, z), magic.Q1(x).

Modified Rules

P1 (x, y) :-
magic.P1(x), B1(x, w, x_1), Q1(x_1, y), B2(w, x_2), Q1(x_2, y), B3(y, z).

Q1 (x, y) :- magic.Q1 (x), B4(x, z), P1(z, y).

P1 (x, y) :- magic.P1(x), B5(x, y).

Figure 6. *The Magic Set Method for Francois' Example*

plan having some desirable performance characteristics. First of all, the magic set rules and modified rules can be generated efficiently:

PROPOSITION 2. *Let* $Q = <G, LP, D>$ *be a query that has the binding passing property and let there be a bound on the arity of the predicates in* LP. *Then the general magic set method constructs the modified query* $\overline{Q} = <\overline{G}, \overline{LP}, D>$ *in time linear in the size of* LP.

PROOF. There are at most two rules for each arc in the binding graph. By Proposition 1 the number of arcs is $O(s+g)$, where s is the size of LP and g is the size of G. Furthermore, it is easy to see that a recursive or modified rule can be constructed while reading the rule. Hence the modified query is constructed in time linear in the size of LP and G. □

The performance of the magic set method versus alternate approaches has been studied in [BR], where it was shown that it is clearly superior to the fixpoint computation, but not as efficient as the generalized counting method [SZ2] and the well-known method of [HN] when duplicates do not occur in the generation of the magic sets and/or of the answer. The magic set method is however better at handling duplicates and also has the important property that it guarantees safety in situations where other methods do not (see discussion in Section 7). An improvement of the magic set method designed to enhance its performance is discussed in the next section.

5. The Minimagic Method

The magic set method tends to compute the solved predicates twice. Once in the computation of magic sets, and a second time in the final fixpoint computation. For example in Figure 5 the $\geq$ and $<$ predicates are repeated in the Magic Rules and modified Rules.

These problems can be solved by using two sets of specialized magic predicates, as follows:

(i) The *supplementary magic predicates* record useful results obtained during the top down phase, and are thus used in the recursive modified rules.

(ii) The *minimagic predicates* are derived from the above and used in the generation of modified exit rules.

Therefore the minimagic method transform query $Q = < G, LP, D >$ that has the binding passing property into a query $\overline{Q} = < \overline{G}, \overline{LP}, D >$ which has the same answer as Q

5.1. Minimagic and Supplementary Magic Rules

Minimagic Exit Rule:

The first minimagic rule is generated by the source node in M_Q, say P^S which corresponds to the query goal. Say that the query goal has $n = |S| \geq 1$ bound arguments with respective values $a_1, \cdots, a_n$. Then we add the following clause:

$$minimagic.P^S(a_1, \cdots, a_n).$$

Supplementary Variables:

Given a rule r_i and its head predicate bindings S, the supplementary variable set contain all the *bound variables* that are to become useful in the second (bottom up) phase, thus including:

V_H: the variables appearing in the head,

V_U: the variables appearing in some unsolved datum predicate,

V_C: the variables appearing in some c-predicate of the body.

If V_B denote the set of all bound variables in r_i, then the set of supplementary variables is:

$$V_{sp} = V_B \bigcap (V_H \bigcup V_U \bigcup V_C)$$

Thus, V_{sp} contains all solved variables but those that only appear in solved datum predicates. For instance in the rule r_0 of Figure 6, the variable w will be left out of V_{sp}.

Supplementary Magic Rules:

There is a *supplementary magic rule* for each node R^S in M_Q and each bundle of arcs labeled with the same rule name, say r_i, leaving R^S (thus r_i has R as head predicate symbol).

$$supmagic.R^S.r_i(y_1, \cdots, y_l) :-$$
$$minimagic.R^S(x_1, \cdots, x_n),\ Q_1, \cdots, Q_h$$

where:

i) $x_1, \ldots, x_n$ are the bound arguments in the head of r_i (i.e., those in S),

ii) $Q_1, \cdots, Q_h$ are the datum predicates of r_i, solved by the variables in the bound arguments S.

iii) $y_1, \cdots, y_l$ are the supplementary variables.

Minimagic Rules:

The minimagic sets are computed from the set of supplementary magic predicates. There is a minimagic rule for each arc in M_Q. For an arc from node R^S to node P^T, with label $[r_i, j]$ we add a rule

$$minimagic.P^T(z_1, \cdots, z_k) :- \ supmagic.R^S.r_i(y1, \cdots, y_l)$$

where:

i) $z_1, \cdots, z_k$ are the bound arguments in the j-th predicate of r_i (i.e., those in T), while

ii) $y_1, \cdots, y_l$ are the supplementary variables for rule r_i with head bindings S.

5.2. Modified Rules for the Minimagic Method

Modified Recursive Rules:

A number of new predicate symbols are introduced, one for each node in M_Q, to replace the c-predicate symbols in LP. For each node in M_Q, there are as many modified rules as there are bundles of arcs from the node labeled with the same rule name. Thus, let R^S be a node in M_Q and r_i be the label of a bundle of arcs leaving R^S (thus r_i is a recursive rule with head predicate symbol R); then, the original rule r_i is replaced by:

$$R^S(x_1, \cdots, x_n) :- \\ supmagic.R^S.r_i(y_1, \cdots, y_1), P_1, \cdots, P_k, Q_1, \cdots, Q_h$$

where:

(i) $y_1, \cdots, y_l$ denote the supplementary variables in r_i given that the bound arguments in the head are S.

(ii) $P_1, \cdots, P_k$ are the original c-predicates each adorned by its bound arguments, and

(iii) $Q_1, \cdots, Q_h$ are the unsolved datum predicates.

Modified Exit Rules:

Say that R^S is a node of M_Q and there is an exit rule r_i with head predicate symbol R. Then we modify the original rule r_i by

(i) replacing the predicate symbol R by R^S in the head of the rule, and

(ii) the addition of the goal

$$minimagic.R^S(x_1, \cdots, x_n)$$

to its body, where $x_1, \cdots, x_n$ denote the bound arguments in the head of r_i

Modified Goal:

If the source node in M_Q is P^S then the modified goal becomes $P^S(x1, \cdots, x_n)$ where $x_1, \cdots, x_n$ denote the arguments (bound or unbound) in the query goal.

Figures 7,8,9 and 10 give some examples of the application of the minimagic method.

From a formal viewpoint, the minimagic method has properties similar to those of the magic set method since it preserves the equivalence of queries (as per Theorem 1) and the modified program can be constructed in polynomial time (as per Proposition 2). The minimagic method is similar to the Alexander method [RLK], also discussed in [BeR].

6. Arbitrary Datum Predicates.

As previously mentioned, datum predicates need not be restricted to database and comparison predicates; all is required is that these predicates can be solved independently of the recursive clique under consideration. For instance, the technique presented in [Z3] can be used to deal effectively with *non-recursive rules* possibly containing function symbols. That technique also consists of (i) a binding passing analysis, and (ii) a modified execution plan. Said binding passing analysis is based on a (functional) dependency model that determines what other variables are bound once an initial set is given, using a computationally efficient closure algorithm. (Thus it represent a generalization of binding propagation rules described in

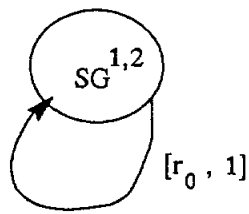
G : SG(a, b)?

LP:

r_0: SG(x, y):- P(x, x_1), P(y, y_1), SG(x_1, y_1).

r_1: SG(x, x):- H(x).

Binding Graph



Minimagic and Supplementary Magic Rules

minimagic.SG1 2(a, b).

supmagic.SG1 2.r_1 (x, y, x_1, y_1) :-
minimagic.SG1 2(x, y), P(x, x1), P(y, y1).

minimagic.SG1 2(x_1, y_1) :- supmagic.SG1 2.r_1(x, y, x_1, y_1).

Modified Rules

SG1 2(x, y) :- supmagic (x, y, x_1, y_1), SG1 2(x_1, y_1).

SG1 2(x, x) :- minimagic.SG1 2(x, x,), H(x).

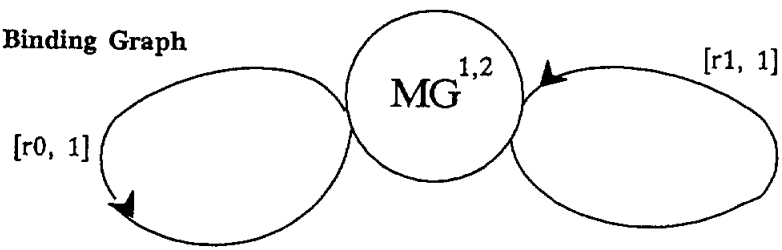
Figure 7. *The Minimagic Set Method for SG(a,b)?*

G : MG(L_1, L_2, W)?

LP:

r0: MG(x•y, x_1•y_1, x•w) :- MG(y, x_1•y_1, w), $x \geq x_1$.
r1: MG(x•y, x_1•y_1, x_1•w) :- MG(x•y, y_1, w), $x < x_1$.
r2: MG(**nil**, x, x).
r3: MG(x, **nil**, x).

Binding Graph



Minimagic and Supplementary Magic Rules

minimagic.MG1 2(L_1, L_2).

supmagic.MG1 2.r_0 (x, y, x_1, y_1) :-
minimagic.MG1 2(x•y, x_1•y_1), $x \geq x_1$.

supmagic.MG1 2.r_1 (x, y, x_1, y_1) :-
minimagic.MG1 2(x•y, x_1•y_1), $x < x_1$.

minimagic.MG1 2(y, x_1•y_1) :- supmagic.MG1 2.r_0 (x, y, x_1, y_1).

minimagic.MG1 2(x•y, y_1) :- supmagic.MG1 2.r_1 (x, y, x_1, y_1).

Modified Rules

MG1 2(x•y, x_1•y_1, x•w) :-
supmagic.MG1 2.r_0 (x, y, x_1, y_1), MG1 2(y, x_1•y_1, w).

MG1 2(x•y, x_1•y_1, x•w) :-
supmagic.MG1 2.r_1 (x, y, x_1, y_1), MG1 2(x•y, y_1, w).

MG1 2(**nil**, x, x) :- minimagic.MG1 2(**nil**, x).

MG1 2(x, **nil**) :- minimagic.MG1 2(x, **nil**).

Figure 8. *Minimagic Set Implementation of the List Merge.*

G : P(a, y)?

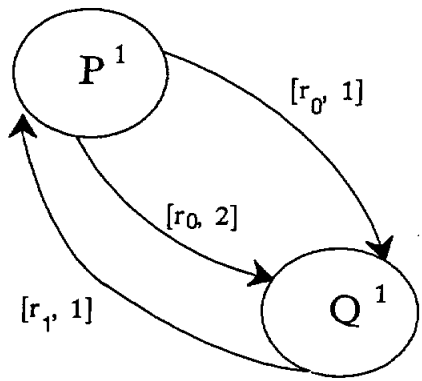
LP :

r_0: P(x, y) :- B1(x, w, x_1), Q(x_1, y), B2(w, x_2), Q(x_2, y), B3(y, z).

r_1: Q(x, y) :- B4(x, z), P(z, y).

r_2: P(x, y) :- B5(x, y).

Binding Graph



Minimagic and Supplementary Magic Rules

minimagic.P1 (a).

supmagic.P1.r_0 (x, y, x_1, x_2) :-
minimagic.P1(x), B1(x, w, x_1), B2(w, x_2).

supmagic.Q1.r_1 (x, z) :- minimagic.Q1(x), B4(x, z).

minimagic.Q1(x_1) :- supmagic.P1.r_0 (x, x_1, x_2).

minimagic.Q1(x_2) :- supmagic.P1.r_0 (x, x_1, x_2).

minimagic.P1(x) :- supmagic.Q1.r_1 (x, z).

Modified Rules

P1(x, y) :- supmagic.P1.r_0(x, x_1, x_2), Q1(x_1, y), Q1(x_2, y), B3(y, z)

Q1(x, y) :- supmagic.Q1.r_1(x, z), P1(z, y).

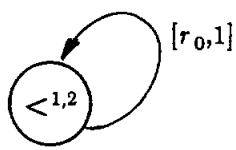
P1(x, y) :- minimagic.P1(x), B5(x, y).

Figure 9. *The Minimagic Set Method for Francois' Example.*

Section 3.1.) Given a set of bound variables in a recursive rule we can therefore use the algorithm in [Z3] to determine the remaining bound variables in *non-recursive* predicates.

Let us consider now the problem of determining whether recursive predicates (not in the same recursive clique as our c-predicates), can be used as solved datum predicates. This tantamounts to determining whether the corresponding goal in the rule can be solved for the given set of bindings. To this end, we can apply the known techniques for solving recursive predicates, in particular the methods described here. Take for instance a query $G: MG(L_1, L_2, X)$, defined against a LP consisting of the rules of Figure 2 and 3 combined. Then, in order to solve this query, we will also have to solve the goal $G2$: $C_1 < C_2$, where C_1 and C_2 stand for arbitrary constants. Thus we get the modified set of rules of Figure 10.

G: $C_1 < C_2$?
LP:
r_0: $x < s(x)$.
r_1: $x < s(y) :- x < y$.
Binding Graph:



Minimagic and Supplementary Magic rules:
minimagic. $<^{1,2}(C_1, C_2)$
supmagic. $<^{1,2}.r_1(x, y) :-$ *minimagic*. $<^{1,2}(x, s(y))$
minimagic. $<^{1,2}(x, y) :-$ *supmagic*. $>^{1,2}.r\,1(x, y)$
Modified Rules and Goal:
$<^{1,2}(x, s(y)) :-$ *supmagic*. $<^{1,2}.r\,1(x, s(y)), <^{1,2}(x, y)$.
$<^{1,2}(x, s(x)) :-$ *minimagic*. $<^{1,2}(x, s(x))$.
$\overline{G}$: $<^{1,2}(C_1, C_2)$?

Fig. 10. *Implementation of the "less-than" rules of Figure 3.*

For a complete implementation plan, the goals $x < x_1$ in the rules in Figure 5 and 8, must be given a suitable interpretation that basically links those rules with those of Figure 10. For instance, for Figure 10, said goal could be redefined as follows:

$$x <_{x_1} :- \textit{assert}(\textit{minimagic}. <^{1,2}(x, x_1)), <^{1,2}(x, x_1).$$

7. Safety of Queries

A safe query is one that generates only a finite number of answers. Safety for recursive queries with function symbols is undecidable; thus the best a person can do is to provide sufficient conditions that cover the cases of practical interests. Our domain of interest consists of recursive queries having the binding passing properties for which we want to ensure that the our methods will terminate. Note that the magic set (the minimagic) method recast the original query Q into two fixpoint computations: whenever both these computations terminate in a finite number of steps, we will say that *the magic set method (the minimagic method) is safe w.r.t. to the query* Q.

Thus, the following two properties follow immediately from the definitions:

PROPOSITION 3. *The magic set (the minimagic method) is safe w.r.t. a query having the binding passing property if and only if the magic set (minimagic set) fixpoint computation converges in a finite number of steps.*

PROPOSITION 4. *The magic set method is safe w.r.t. to a query Q having the binding passing property if and only if the minimagic method is safe w.r.t. Q.*

Thus any safety property discussed next, that holds for the magic set method also holds for the minimagic method (and vice versa).

We will now limit our attention to datum predicates are either database predicates or comparison predicates. We can thus state the following sufficient conditions for safety:

PROPOSITION 5. *Let $Q = <G, LP, D>$ be a query having the binding passing property. A sufficient condition for the magic set to be safe with respect to Q is that every recursive rule in LP does not have any function symbol in its body. This condition can be tested in time polynomial in the size of LP.*

Proposition 5 expresses a more general condition for safety than that given in [U2] where the absence of function symbols on the right hand side of rules must be complemented with additional

conditions to guarantee safety.

Using Proposition 5, it is easy to see that the magic set method is safe with respect to the same generation query (Figure 1), the Francois' query (Figures 6 and 9) and the less-than query (Figure 10). On the other hand, this proposition does not guarantee the safety of the magic set method with respect to the list-merge query (Figure 2). Therefore, we need to generalize Proposition 5 as follows. Say that x is a *subcomponent of* y if either $y = x$, or $y = f(x_1, \cdots, x_n)$ with x a subcomponent of x_i for some $1 \leq i \leq n$. Also observe that the body of of each recursive magic rule contains exactly one magic predicate. Thus,

PROPOSITION 6. *Let Q be a query having the binding passing property and $\overline{Q}$ be the modified query according to the magic set method. If for each recursive magic rule $\hat{r}_i$, each complex argument x of the head of $\hat{r}_1$, x is a subcomponent of some argument of the magic predicate in the body of $\hat{r}$, then the magic set method is safe w.r.t. Q. This condition can be tested in time polynomial in the size of $\hat{Q}$.*

Thus, by Proposition 6, the magic set method is safe with respect to the list-merge query (Figure 2). This proposition is very useful for determining the safety of recursive predicates with function symbols, including typical situations, such as appending two lists and searching and manipulating trees and lists. An interesting area of further research is finding more general sufficient conditions for the magic set method to be safe.

8. Conclusion

The numerous methods proposed for supporting recursion in logic based languages were reviewed and their performance compared in a recent study [BR]. This study indicates that the magic counting method is one of the techniques most generally applicable and, in terms of performance, it ranks above most other methods, but below the counting method. The last result has been also confirmed in [MPS]. However, the magic set method remains the method of choice, since it is safe in all cases where the counting method is, and it is also safe in situations where the counting method fails to terminate (e.g., when there are cycles in the database). The minimagic method here introduced improves on the magic set method by

removing redundant work while retaining its advantages in terms of safety and generality. Current research attempts to generalize the magic counting method [SZ1] that merges the magic set (or minimagic) method with the counting method to obtain the strengths of both [SZ3].

Acknowledgments:
The authors are grateful to Isaac Balbin, Danette Chimenti, Michael Kifer, Ravi Krishnamurthy, and the referees for various corrections and improvements, and to François Bancilhon and Raghu Ramakrishnan for many inspiring discussions.

References

[AU] Aho A. V. and J. Ullman, " Universality of Data Retrieval Languages," *Proc. POPL Conference,* San Antonio Tx, 1979.

[Ban] Bancilhon, F., "Naive Evaluation of Recursively defined Relations", *On Knowledge Base Management Systems,* (M. Brodie and J. Mylopoulos, eds.), Springer-Verlag, 1985.

[BaR] Balbin, I., K. Ramamohanarao, "A Differential Approach to Query Optimization in Recursive Deductive Databases", *Journal of Logic Programming,* to appear.

[BeR] Beeri, C. and Ramakrishnan R., "On the Power of Magic," *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* pp. 269-284, 1986.

[BKBR] Beeri, C., P. Kanellakis, F. Bancilhon, R. Ramakrishnan, "Bound on the Propagation of Selection into Logic Programs", *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* pp. 214-226, 1986.

[BNRST] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli and S.Tsur, "Sets and Negation in a Logic Database Language (LDL1)," *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* pp. 21-38, 1986.

[BMSU1] Bancilhon, F., D. Maier, Y. Sagiv, J. Ullman, "Magic sets and other strange ways to implement logic programs", *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* 1986.

[BMSU2] Bancilhon, F., D. Maier, Y. Sagiv, J. Ullman, "Magic sets: algorithms and examples", unpublished manuscript, 1985.

[BR] Bancilhon, F., and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," *Proc. ACM SIGMOD Int. Conference on Management of Data,* Washington, D.C., May 1986.

[CG] Clark K.L. and Gregrory S., "A relational language for parallel programming," *Proc. ACM Conference on Functional Languages and Computer Architecture,* ACM, October, 1981.

[CH] Chandra, A.K., Harel, D., "Horn clauses and the fixpoint hierarchy", *Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* 1982, pp. 158-163.

[CM] Copeland, G. and D. Maier, "Making SMALLTALK a Database System", *Proc. ACM SIGMOD Conference,* pp. 316-325, 1984.

[DKM] Dwork, P., P. Kanellakis and J., Mitchell, "On the Sequential Nature of Unification", *Journal of Logic Programming, Vol 1, pp. 35-50, 1984.*

[GD] Gardarin, G. and C. DeMaindreville, " Evaluation of Database Recursive Logic Programs as Recursive Function Series," *Proc. ACM SIGMOD Int. Conference on Management of Data,* Washington, D.C., May 1986.

[GMN] Gallaire, H.,J. Minker and J.M. Nicolas,"Logic and Databases: a Deductive Approach," *Computer Surveys,* Vol. 16, No. 2, 1984.

[HN] Henschen, L.J., Naqvi, S. A., "On compiling queries in recursive first-order databases", *JACM 31,* 1, 1984, pp. 47-85.

[KL] Kifer, M. and Lozinskii, E.L., "Filtering Data Flow in Deductive Databases," *ICDT'86,* Rome, Sept. 8-10, 1986.

[KT] Kellog, C., and Travis, L., "Reasoning with data in a deductively augmented database system", in *Advances in Logic and Databases, Vol. 1* (Gallaire, H., Minker, J., eds), Plenum, New York, 1981.

[KY] Kunifji S., H. Yokota, "Prolog and Relational Databases for 5th Generation Computer Systems," in *Advances in Logic and Databases, Vol. 2*

[Llo] Lloyd, J. W., *Foundations of Logic Programming,* Springer Verlag, 1984.

[MK] Maluszynski, J. and H.J. Komorowski, "Unification-free Execution of Logic Programs", *Procs. 1985 Symposium on Logic Programming,* IEEE, 1985.

[MPS] Marchetti-Spaccamela, A., Pelaggi, A., Saccà, D., "Worst-case complexity analysis of methods for logic query implementation", 1986, *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* pp. 294-301, 1986.

[Nai] Naish, L., "Automating Control for Logic Programs", *The Journal of Logic Programming,* 1985.

[Nil] Nilson, N.J. "*Principles of of Artificial Intelligence*", Tioga Publ. Co., Palo Alto, Calif., 1980.

[Par] Parker, S. et al., "Logic Programming and Databases," in *Expert Database Systems,* L. Kerschberg (ed.), Benjamin/Cummings, 1986.

[Rei] Reiter, R., "On closed world databases", in *Logic and Databases* (Gallaire, H., Minker, J., eds), Plenum, New York, pp. 55-76, 1978.

[RLK] Rohmer, J., R. Lescouer and J.M. Kerisit, "The Alexander Method – A Technique for the Processing of Recursive Axioms in in Deductive Databases," *New Generation Computing,* Vol. 4, No. 3, pp. 273-287, 1986.

[SG] Smith, D. and Genesereth, M.R. "ordering Conjunctive Queries", *Artificial Intelligence,* 26, 171-215, 1985.

[ST] Shapiro, E. and A. Takeuchi, "Object-Oriented Programming in Concurrent Prolog," *Journal of New Generation Computing, Vol.1, pp. 25-49, 1983.*

[SZ1] Saccà, D., Zaniolo, C., "On the implementation of a simple class of logic queries for databases", *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* 1986.

[SZ2] Saccà, D., Zaniolo, C., "The Generalized Counting Method for Recursive Logic Queries," *ICDT '86 Proceedings,* 1986.

[Tar] Tarski, A. "A Lattice Theoretical Fixpoint Theorem and its Application," *Pacific Journal of Mathematics No. 5,* pp. 285-309, 1955.

[TZ] Tsur, S. and C. Zaniolo, "LDL: A Logic-Based Data Language," *Proc. of 12th VLDB,* Tokyo, Japan, 1986.

[U1] Ullman, J.D., *Principles of Database Systems,* Computer Science Press, Rockville, Md., 1982.

[U2] Ullman, J.D., "Implementation of logical query languages for databases", *TODS 10,* 3, 1985, pp. 289-321.

[UV] Ullman, J.D. and A. Van Gelder, "Testing Applicability of Top-Down Capture Rules," Stanford University, Report STAN-CS-85-1046, 1985.

[VK] van Emden, M.H., Kowalski, R., "The semantics of predicate logic as a programming language", *JACM 23,* 4, 1976, pp. 733-742.

[Vg] Van Gelder, A., "A Message Passing Framework for Logical Query Evaluation," *Proc. ACM SIGMOD Int. Conference on Management of Data,* Washington, D.C., May 1986.

[Vi] Vieille, L. "Recursive Axioms in Deductive Databases: the Query-Subquery Approach," *Proc. First Int. Conference on Expert Database Systems,* Charleston, S.C., 1986.

[Z1] Zaniolo, C. "Prolog: a Database Query Language for all Seasons," in *Expert Database Systems,* L. Kerschberg (ed.), Benjamin/Cummings, 1986.

[Z2] Zaniolo, C. "The Representation and Deductive Retrieval of Complex Objects," *Proc. 11-th VLDB,* pp. 459-469, 1985.

[Z3] Zaniolo, C. "Safety and Compilation of Non-Recursive Horn Clauses," *Proc. First Int. Conference on Expert Database Systems,* Charleston, S.C., 1986.