The Generalized Counting Method for Recursive Logic Queries

Domenico Saccà †

CRAI, Rende, Italy

Carlo Zaniolo

MCC, Austin, Texas, USA

ABSTRACT

This paper treats the problem of implementing efficiently recursive Horn Clauses queries, 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 the recursive query. A general method is given for solving these queries; the method 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. The first such computation emulates the propagation of bindings in the top-down phase; the second generates the desired answer by proving the goals left unsolved during the previous step. Finally, sufficient conditions for safety are derived, to ensure that the fixpoint computations are completed in a finite number of steps.

1. Introduction

This work is motivated by the belief that an integration of technologies of Logic Programming and Databases is highly desirable, and will supply a corner stone of future Knowledge Based Systems [P, U2]. Prolog represents a powerful query language for database systems, and can also be used as a general-purpose language for application development, particularly in the symbolic manipulation and expert system areas [Z1]. However, Prolog's sequential execution model and the spurious non-logical constructs thus grafted on the language constitute serious drawbacks for database applications, since

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

i) they imply a one-tuple-at-the-time, nested-loop join strategy which is not well-suited for parallel processing, and tends to be inefficient when the fact base is stored on disk, and

ii) the programmer must guarantee the performance and the termination of the program by carefully ordering rules and goals— a policy that limits the ease-of-use of the language and the data independence of applications written in it.

Thus, we should move beyond Prolog, to a pure Logic-based language amenable to secondary storage and parallel implementation, where the system assumes responsibility for efficient execution of correct programs — an evolution similar to that of databases from early navigational systems to relational ones. Towards this ambitions objective, we take the approach of compiling the intentional information expressed as Horn clauses and queries into set-oriented processing primitives, such as relational algebra, to be executed on the extensional database (fact base). This is a simple process for Horn Clauses containing only non-recursive predicates with simple variables, inasmuch as these rules basically correspond to the derived join-select-project views of relational databases [U1]. Horn clauses, however, contain two powerful constructs not found in the Relational calculus: one is recursion, that, e.g., entails the computation of closures, the other is general unification that, via the use of function symbols, can be used to support complex and flexible structures (not just flat tuples as is relational databases). The efficient implementation of these two powerful constructs poses some interesting problems [HN, MS, CH, U2, L, BMSU1, SZ1, Z2, GD, Vg, Vi]. For instance, the technique of using the query constants to search the database efficiently (pushing selection) is frequently inapplicable to recursive predicates [AhUl]. Moreover, the issue of safety, which in relational databases is solved by simple syntactic conditions on the query language, here requires a complex analysis on the bindings passed upon unification [UV,Z3].

This paper studies the problem of implementing safely and efficiently recursive Horn clauses in the presence of query constants and thus it introduces a powerful technique, called *the generalized counting method,* that is more effective in dealing with recursive predicates with function symbols than those previously known [HN, U2, BMSU1, BMSU2, SZ1].

2. Fixpoint Evaluation of Recursive Queries

Take the recursive rule 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[1]. Then, a query such as:

G1: $SG(x,y)$*?*

defines all persons that are of the same generation. The answer to this query can be computed as the least fixpoint of the following function over relations:

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

Our function f is defined by a relational algebra expression having as operands the constants H and P and the variable SG (H and P denote the database relations with respective function symbols H and P and respective arities one and two — whereas SG is an unknown relation with arity two). Therefore, the computation of the least fixpoint can proceed by setting the initial value of SG to the empty set and computing $f(SG)$. Then $f(SG)$ becomes the new value for SG and this iterative step is repeated until no more tuples can be added to SG, which then becomes the answer to the query. Since all goals in a Horn clause are positive, the corresponding relational expression is monotonic, w.r.t. the ordering on relations defined by set containment. Thus, there exists a unique least fixpoint [T1]. The fixpoint computation approach, refined with the differential techniques, such as those described in [B, BGK], supplies an efficient algorithm for implementing queries with no bound argument. This approach, however, becomes inefficient for common queries, such as $G2$ below, where arguments are either constant or otherwise bound to a small set of elements:

G2: $SG(john,y)$*?*

This query retrieves all humans of the same generation as "john". A naive application of the fixpoint approach here implies generating all possible pairs of humans in the same generation, to discard then all but those having "john" as their first component. Much more efficient strategies are possible; Prolog's backward chaining, for instance, propagates all the bindings downwards, during the top-down phase (from the goal to database), then, during the bottom-up phase, performs a fixpoint computation using only those database facts that were found

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

relevant in the previous phase. (For the example at hand, the only relevant facts are those describing ancestors of "john".) In traditional databases this top-down binding propagation strategy corresponds to the well-known optimization technique of pushing selection inside the relational algebra expressions. We need here to extend and generalize this technique to the case of recursive predicates. The importance of the problem considered is underscored by the safety issue for "computational" predicates, such as the merge example of Figure 2, which are normally defined using both recursion and function symbols.

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.*

The problem of supporting *non-recursive* Horn clauses with function symbols was studied in [Z2]. Since predicates have structured arguments (for instance the first argument in the the head of r_1 in Figure 2 has x and y as subarguments), an Extended Relational Algebra (ERA) was proposed in [Z2] to deal with them. A first operator, called *extended select-project,* entails the selection of subcomponents in complex arguments (in this particular case where the dot is our (infix) function symbol, this operator performs "car" and "cdr" operations on dotted lists). The second operator, called a *combine,* allows one to build complex arguments from simpler ones (on a dotted list, this corresponds to the "cons" operator). Non-recursive Horn clauses can be implemented as ERA expressions [Z2, Z3]. Moreover, since functions defined using ERA expressions are still monotonic, the basic fixpoint computation approach (bottom-up execution) remains applicable to predicates with function symbols.

However, there are safety issues which limit the applicability of the fixpoint approach, since Herbrand's universe is infinite when function symbols are involved. For instance, the relations representing all possible sets of values for x and x_1 in our rules of Figure 2, are infinite; furthermore even if we restrict these variables to a finite set, rules r_1 and r_2 would generate longer and longer lists at each step of the fixpoint computation, which therefore becomes a non-terminating one.

In reality, the potential safety problem previously described are avoided because a procedure, such as that of Figure 2, is only invoked as a goal with certain arguments bound, to derive the the unbound ones. Typically for instance, the first two arguments are given to derive the third one (or perhaps, the third argument is given to generate all the pairs of lists that merge into this one, or some combination of these two situations). This observation also extends to predicates without function symbols; for instance, the same-generation example might be written without the $H(x)$ goal in the second rule of Figure 1, because of an implicit assumption that SG will only be called with some arguments bound — an assumption that better be verified before an relational algebra equivalent can be generated for the SG rules.

In conclusion, an effective usage of the binding information available during the top-down phase is vital for performance reasons and to avoid the non-termination pitfall. The purpose of this paper is to propose a general framework and efficient algorithms to deal with this problem. The basic approach consists of the 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 computation of special sets (i.e., the counting sets and the supplementary counting sets) that actually implement the top-down propagation of bound values,

iii) a modified bottom-up computation that generates the values satisfied by the queries.

The method presented here, is more powerful than methods previously proposed in the literature with respect to the treatment of recursive predicates with function symbols. For instance, the queries on the MG example of Figure 2, can not be handled with the methods proposed in [U2] that doe not allow for function symbols on the right side of rules.

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 $G2{:}SG(john, x)?$ 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 bound then all the other variables are bound.

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

Let P be a predicate in the body of r_i. Then, an *argument* of P will be said to be *bound* when all its variables are bound.

Say that S denotes the bound arguments in the head predicate of r_i and B the (bound) variables in these arguments, moreover, say that T denotes 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.*

Solved Predicates:

A datum predicate of r_i will be said to be *solved* when all its variables are bound.

Say for instance that the first argument of SG is bound in Figure 1. Then x is bound and so is x_1 (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 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 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.

Let r_i be a rule and S be the set of the 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 $B_S \bigcup B_c$, where:

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

B_c are the variables of c-predicates in the body of r_i.

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 rule r_i such that the predicate symbol of its head is R, is *solved* by S.

Thus our examples in Figures 1 and 2, with binding graphs of Figure 4 and 5, have the binding passing property. This property guarantees that (a) 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 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 is propagated through an argument of a c-predicate only if the whole argument is bound. A more detailed analysis could consider that the binding can be also passed through partially bound arguments of c-predicates. 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 [SZ3].

PROPOSITION 1. *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 polynomial in the size of LP.*

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

4. The Generalized Counting Method.

We now present a method to implement logic queries which have the binding passing property defined in the previous section. This method, called the *generalized counting method*, is an extension of the counting method described in [SZ1] for solving a particular class of logic queries without function symbols and without comparison predicates. An informal description of the counting method was first given in [BMSU1].

The generalized counting method recasts a query that is inefficient or unsafe to compute in a single fixpoint computation, into a pair of safe and efficient fixpoint computations. While this pair could be expressed directly in terms of relational algebra [SZ1], reasons of simplicity, expressivity and independence from the target implementation language suggest to represent it by recursive rules. Thus the generalized counting method can be viewed as a rule rewriting system that maps a query $Q = < G, LP, D >$ into an equivalent query $\overline{Q} = < \overline{G}, \overline{LP}, D >$ that can be computed safely and efficiently using the fixpoint approach described in Section 2. In $\overline{LP}$ we find two new sets of rules, called *counting rules* and *supplementary counting rules*, that perform the top-down propagation of bound values; in addition, we find every rule of LP transformed into one or more rules (*modified rules*) that perform the bottom-up computation of the final anwer.

4.1. Counting and Supplementary Counting Rules

The overall translation process consists of (i) the generation of counting and supplementary counting sets to perform the top-down propagation of bound values, and (ii) the modification of the original goal and rules to take advantage of the counting sets.

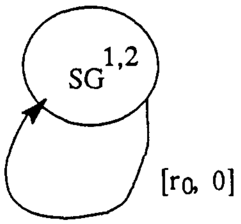
To generate counting sets, a number of new predicate symbols are introduced, one for each node of the binding graph M_Q of Q. Thus, for each node R^S we introduce a new predicate $cnt.R^S$ with $| S | + 3$ arguments. Thus, there is an argument for each bound argument in the head of the original rule, plus three additional integer arguments respectively recording (i) the level of the recursive call, (ii) the recursive rule used and (iii) the c-predicate occurrence used in the

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



Counting Rules:

cnt.$SG^{1,2}$ (0, 0, 0, a, b).

cnt.$SG^{1,2}$ (j+1, 1*k+0, 1*h+0, x_1, y_1):- cnt.$SG^{1,2}$ (j, k, h, x, y), P(x, x_1), P(y, y_1).

Supplementary Rules: None

Modified Rules and Goal:

$SG^{1,2}$ (j–1, (k–0)/1, h/1) :-$SG^{1,2}$ (j, k, h).

$SG^{1,2}$ (j, k, h) :- cnt.$SG^{1,2}$ (j, k, h, x, x), H(x).

$\overline{G}$: $SG^{1,2}$ (0, 0, 0)?

Figure 4. The Same Generation Example for G: SG(a,b)?

body of the rule.

Exit Counting Rules:

The first counting 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 for the counting set:

$$cnt.P^S(0,0,0,a_1, \cdots, a_n).$$

Recursive Counting Rules:

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

$$cnt.P^T(j+1, m \times k+i, p \times h+v, y_1, \cdots, y_l) :- \quad cnt.R^S(j, k, h, x_1, \cdots, x_n),\ Q_1, \cdots, Q_q$$

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 v-th c-predicate of r_i (i.e., those in T),

iii) $Q_1, \cdots, Q_q$ are the predicates of r_i, solved for bound arguments S.

iv) j, k and h are the running indices, while m and p are constants characterized as follows:
m is the total number of recursive rules,
p denotes the total number of c-predicates in the body of r_i.

It should be clear, that in the rule above we have taken liberties with the notation by representing operations on indexes directly by their arithmetic expression rather than introducing new goals, such as "j' is $j+1$ ", "k' is $m \times k+i$" and "h' is $p \times h+s$ ", and then writing the head as $cnt.P^T(j', k', h', y_1, \cdots, y_l)$, as required, say, in Prolog. But we have used this more concise notation since it is suggestive of the counting operations to be performed during the fixpoint computation.

Informally described, the counting rules are constructed by eliminating all unbound arguments and unsolved datum predicates, exchanging the a c-predicate in the body with that in head, and adding the two indexes. Note that, while there are as many counting rules as arcs in the graph, there are only as many counting predicates as there are nodes — see Figure 5 for an example.

In the top-down generation of the counting sets we often generate values that are needed in the successive bottom-up computation. For instance, in the merge example of Figure 2, we generate the values of x in rule r_0 and those of x_1 in r_1;

since these will not be part of the c-predicates in the modified rules, they must be saved for later use in the bottom-up phase. The supplementary rules provide the means to this end.

Supplementary Counting Variables:

Consider a node R^S in M_Q. In M_Q, there is arc labeled $[r_i, v]$ out of R_S for each occurrence of a c-predicate in the body of r_i. Say that

a) B^+ denotes the set of the variables bound in r_i by the (bound) variables in the S-arguments of the head of r_i,

b) V_U denotes the set of all variables either appearing in unsolved predicates of r_i, or in unbound arguments of the head of r_i (i.e., those not in S)

c) V_C the set of variables appearing in some unbound arguments of a c-predicate in the body of r_i,

then we need to save the values of every variable in $V_U \cap B^+$, but those in V_C whose values are recomputed in the bottom-up computation anyway (see modified rules). Thus we need to keep the values of all variables in $V_{sp} = (B^+ \cap V_U) - V_C$, which will be called the set of *supplementary counting variables.* For rules where V_{sp} is empty there is no need for a supplementary counting set. Such is the case for the SG example of Figure 4.

Consider however the example of Figure 5. For rule r_0 we have $B^+ = \{x, y, x_1, y_1\}$, while $V_U = \{x\}$. Since $V_C = \{w\}$, the set of supplementary counting variables is $V_{sp} = \{x\}$. Likewise for r_1, the only supplementary counting variable is x_1.

Supplementary Counting Rules:

Then we can add the supplementary counting rules, one for each bundle of arcs labeled with the same rule out of a node for which the set of supplementary counting variables is not empty. If r_i is the rule labeling a bundle leaving, say, node R^S, then, using the counting rule for R^S, we write:

$$spcnt.r_i.R^S(j, k, h, z_1, \cdots, z_t) :- \; cnt.R^S\;(j, k, h, x_1, \cdots, x_n),\;\; Q_1, \cdots, Q_q$$

where,

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

ii) $z_1, \cdots, z_t$ are the supplementary counting variables.

iii) $Q_1, \cdots, Q_q$ are the predicates of r_i, solved for bound arguments S.

For an example see Figure 5.

G : $MG(L_1, L_2, W)$?

LP:

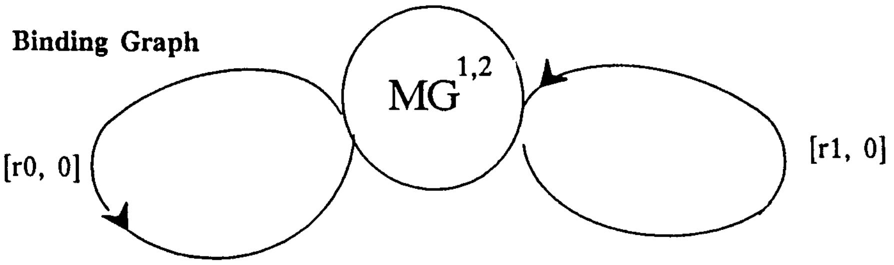
r0: $MG(x{\bullet}y,\ x_1{\bullet}y_1,\ x{\bullet}w)$:- $MG(y,\ x_1{\bullet}y_1,\ w),\ x \geq x_1$.

r1: $MG(x{\bullet}y,\ x_1{\bullet}y_1,\ x_1{\bullet}w)$:- $MG(x{\bullet}y,\ y_1,\ w),\ x < x_1$.

r2: MG(**nil**, x, x).

r3: MG(x, **nil**, x).

Binding Graph



Counting Rules:

cnt.$MG^{1,2}$ $(0,\ 0,\ 0,\ L_1,\ L_2)$.

cnt.$MG^{1,2}$ $(j+1,\ 2{*}k+0,\ 1{*}h+0,\ y,\ x_1{\bullet}y_1)$:- cnt.$MG^{1,2}$ $(j,\ k,\ h,\ x{\bullet}y,\ x_1{\bullet}y_1),\ x \geq x_1$.

cnt.$MG^{1,2}$ $(j+1,\ 2{*}k+1,\ 1{*}h+0,\ x{\bullet}y,\ y_1)$:- cnt.$MG^{1,2}$ $(j,\ k,\ h,\ x{\bullet}y,\ x_1{\bullet}y1),\ x < x_1$.

Supplementary Counting Rule:

spcnt.$MG^{1,2}$.r0 $(j,\ k,\ h,\ x)$:- cnt.$MG^{1,2}$ $(j,\ k,\ h,\ x{\bullet}y,\ x_1{\bullet}y_1),\ x \geq x_1$.

spcnt.$MG^{1,2}$.r1 $(j,\ k,\ h,\ x_1)$:- cnt.$MG^{1,2}$ $(j,\ k,\ h,\ x{\bullet}y,\ x_1{\bullet}y_1),\ x < x_1$.

Modified Rules and Goal:

$MG^{1,2}$ $(j-1,\ (k-0)/2,\ h/1,\ x{\bullet}w)$:- spcnt.$MG^{1,2}$.ro$(j-1,\ k/2,\ h,\ x)$, $MG^{1,2}(j,\ k,\ h,\ w)$.

$MG^{1,2}$ $(j-1,\ (k-1)/2,\ h/1,\ x_1{\bullet}w)$:- spcnt.$MG^{1,2}$.r1$(j-1,\ (k-1)/2,\ h,\ x_1)$, $MG^{1,2}(j,\ k,\ h,\ w)$

$MG^{1,2}$ (j, k, h, x) :- cnt.$MG^{1,2}$ (j, k, h, **nil**, x).

$MG^{1,2}$ (j, k, h, x) :- cnt.$MG^{1,2}$ (j, k, h, x, **nil**).

$\overline{G}$: $MG^{1,2}$ (0, 0, 0, w)?

Figure 5. The List Merge Example

4.2. Modified Rules

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. Thus, let R^S be a node in M_Q and r_i be the label of a bundle of arcs leaving R^S; then, we introduce the following rule (again we take liberties with the notation by denoting with $k-i/m$ an integer division that succeds only if the remainder is zero):

$$R^S(j-1, (k-i)/m, h/p, u_1, \cdots, u_n) :-$$
$$spcnt.R^S.r_i(j-1,(k-i)/m, h/p, z_1, \cdots, z_t),\ \hat{P}_0, \cdots, \hat{P}_{p-1},\ W_1, \cdots, W_q.$$

where:

i) $u_1, \ldots, u_n$ are the unbound arguments in the head of r_i (i.e., those that do NOT belong to S),

ii) $spcnt.R^S.r_i(j-1,(k-i)/m, h/p, z_1, \cdots, z_t)$ is the supplementary counting predicate, if any, with $z_1, \cdots, z_t$, $t>0$ supplementary counting variables.

iii) $W_1, \cdots, W_q$ are the predicates of r_i, NOT solved for bound arguments S.

iv) $\hat{P}_0, \cdots, \hat{P}_{p-1}$ are the modified c-predicate occurrences in the body of r_i, constructed as follows. Say that there is an arc labeled $[r_i, v]$ from R^S to P^T, and $x_1, \cdots, x_l$ are the unbound arguments in the v-th predicate of r_i (i.e., those NOT in T), then

$$\hat{P}_v = P^T(j, k, h+v, x_1, \cdots, x_l).$$

v) j, k and h are the running indices, while m, i and p are constants respectively denoting the total number of recursive rules, the index of the rule labeling the arc, and the total number of c-predicates in the body of this rule. These indexing operations reverse those performed when building the counting sets.

In other words, one has to take the original rule r_i, eliminate all solved variables and bound predicates, add the three indexes (after suitable indexing operations) in each c-predicate, and, finally, add the the supplementary counting predicates, if any. In addition to the modified recursive rules so generated we need some modified exit rules.

Modified Exit Rules:

Say that R^S is a node of M_Q and there is each exit rule r_i with head predicate R. Then we add the following modified exit rule:

$$R^S(j, k, h, u_1, \cdots, u_n) :- \ cnt.R^S(j, k, h, x_1, \cdots, x_l),\ W_1, \cdots, W_q.$$

where:

i) $u_1, \ldots, u_n$ are the unbound arguments in the head of r_i (i.e, those which do NOT belong to S),

ii) $cnt.R^S(j,k,h,x_1, \cdots, x_l)$ is the counting predicate, with $x_1, \cdots, x_l$ the bound arguments in r_i's head.

iii) $W_1, \cdots, W_q$ are the predicates in the body of r_i.

Thus the exit rules are generated by replacing the bound predicates in the head by the indexes and then adding the counting set to the body of the rule.

Modified Goal:

If $\hat{P}$ is the original query goal with bound arguments S and predicate symbol P, then let $x_1, \cdots, x_n$ denote the unbound arguments (i.e., those not in S). Then, the modified query goal is

$$P(0,0,0, x_1, \cdots, x_n)?$$

It thus follow that the fixpoint computation of the modified rules should be stopped after zero values are generated for the indices.

Figure 6 illustrates the application of the method to the a situation involving mutually recursive predicates and more than one c-predicate in the body of a recursive rule (non-linear rule).

4.3. Properties of the Generalized Counting Method.

From a formal viewpoint the Generalized Counting 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 [SZ3]:

THEOREM 1. *Let $Q = <G, LP, D>$ be a query that has the binding passing property. If $\overline{Q} = <\overline{G}, \overline{LP}, D>$ denotes the modified query produced by the generalized counting method, then Q and $\overline{Q}$ compute the same answer.*

(Note however, that in the theorem above, the answer to both queries could be infinite, a problem treated in Section 5.)

From a computational viewpoint, the generalized counting rules prescribe an abstract computation plan having some desirable performance characteristics. First of all, counting and modified rules can be generated efficiently:

PROPOSITION 2. *Let $Q = <G, LP, D>$ be a query such that Q has the binding passing property and there is a bound on the arity of the predicates in LP. Then the generalized counting method constructs the modified query*

G : P(a, y)?

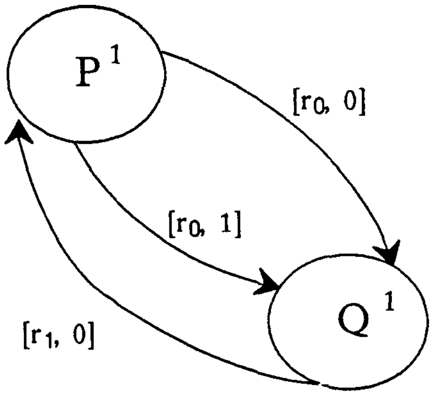
LP : r_0: P(x, y) :- B1(x, x_1), Q(x_1, y), B2(x, x_2), Q(x_2, y), B3(y, z).

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

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

A recursive clique of two mutually recursive predicates: P and Q.

Binding Graph:



Counting Rules:

cnt.P1 (0, 0, 0, a).

cnt.Q1 (j+1, 2*k+0, 2*h+0, x_1):- cnt.P1 (j, k, h, x_1), B1(x, x_1), B2(x, x_2).(from [r_0, 0]

cnt.Q1 (j+1, 2*k+0, 2*h+1, x_2):- cnt.P1 (j, k, h, x), B1(x, x_1), B2(x, x_2). (from [r_0, 1]

cnt.P1 (j+1, 2*k+1, 1*h+0, z):- cnt.Q1 (j, k, h, x), B4(x, z). (from [r_1, 0]

Supplementary Rules: None

Modified Rules and Goal:

P1 (j-1, (k-0)/2, h/2, y):- Q1 (j, k, h, y), Q1 (j, k, h+1, y), B3(y,z).

Q1 (j-1, (k-1)/2, h/1, y):- P1 (j, k, h, y).

P1 (j, k, h, y):- cnt.P1 (j, k, h, x), B5(x, y).

$\overline{G}$: P1 (0, 0, 0, y)?

Figure 6. François' Example

$\overline{Q} = < \overline{G}, \overline{LP}, D >$ *in time polynomial in the size of* LP.

As today, we still lack a general framework that allows us to characterize the performance of the various methods proposed for the compilation of recursive predicates. However, a clear understanding of the behavior of these methods has emerged from the study of typical examples [BR]. These examples strongly suggest that the counting method is superior to the others (in terms of database accesses and computational steps required), particularly in situations that do not require the elimination of duplicates. Thus, the method is ideally suited for situations involving function symbols, where a new term is generated at each step in the fixpoint computations (either by adding some level of nesting in the structure or by removing some). Recursive predicates such as appending two lists, extracting all the elements of a list, searching and manipulating tree structures, etc., are ideal candidates for the generalized counting method.

Our confidence in the ability of the generalized counting method to deal with recursive predicates with function symbols is reinforced by the authors' experience with Prolog and the observation that the generalized counting can be implemented to emulate Prolog very closely. To illustrate this point let us consider the two fixpoint computation prescribed by the generalized counting method. A possible implementation strategy consists of computing all counting set and supplementary counting sets values, before going into the fixpoint computation of the modified rules (a strategy similar to that used in implementing magic sets [BMSU1, BMSU2, SZ2]). However, a modified exit rule with a certain index value, can be fired as soon as the counting set value for that particular index value is obtained. Assuming that no duplicate elimination is needed, the overall strategy then becomes quite similar to that of Prolog (and also to that of [HN]). However, the generalized counting method also allows for massive joins since it does not imply a one-tuple-at-the-time join strategy, and the top-down binding propagation is independent from the ordering of rules and goals.

4.4. Simplifications and Extensions.

A number of simplifications of the overall generalized counting method can be introduced to deal with various subcases.

Single Recursive Rule:

When there is a single recursive rule, the second index remains constant and can be eliminated (see for instance Figure 4).

Single c-predicate in the rule bodies:

When there is a single c-predicate in the body of every rule, the third index remains constant and can be eliminated (see Figure 4).

Shared Solved Predicates

Counting rules and supplementary counting rules might share the same solved predicates. For instance, in Figure 5, the comparison predicates are evaluated in both the counting rules and in the supplementary counting ones; this duplicate work could be eliminated. A general solution to this problem consists in introducing an *allcnt* predicate that computes both the bound arguments and the supplementary counting variables [SZ3]. Then, the counting and special counting predicates can simply be derived from the *allcnt* by projecting out variables not needed in the specific case.

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. Said technique provide a a generalization of the binding propagation rules described in Section 3.1.

Let us now turn to 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 generalized counting method 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 7 (since we only have one recursive rule we only use one index).

Finally, we need to link the rules of Figure 7 with the last counting rule of Figure 5. This can, for instance, be accomplished by redefining the goal $x < x_1$ of Figure 5 as follows:

$$x < s_1 \text{:- } assert(cnt. <^{1,2}(x, x_1)),\ <^{1,2}(0).$$

(This is a rather coarse solution, presented here only as a quick illustration on how things could function; more refined solutions will be given in future reports.)

Trivial Modified Rules

It is easy to see that the only function of the modified recursive rule in Figure 7, is to decrement the index to zero one step at the time. We can thus dispense with this rule and write a new modified goal:

$$\overline{G}: cnt. <^{1,2}(\_, x, s(x))?$$

We have thus eliminated the second fixpoint computation (tail recursion); moreover, we can also drop the index from the counting set computation.

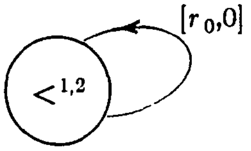
G: C1<C2?

LP:

r_0: $x < s(y)$:- $x < y$.

r_1: $x < s(x)$.

Binding Graph



Counting rules:

cnt. $<^{1,2}(0, C1, C2)$

cnt. $<^{1,2}(j+1, x, y)$:- *cnt.* $<^{1,2}(j, x, s(y))$

Supplementary Counting rules: None.

Modified Rules and Goal:

$<^{1,2}(j)$:- *cnt.* $<^{1,2}(j, x, s(x))$.

$<^{1,2}(j-1)$:- $<^{1,2}(j)$.

$\overline{G}$: $<^{1,2}(0)$?

Fig. 7. *Implementation of the "less-than" rules of Figure 3.*

Symmetrically, it easy to identify many situations where the counting set computation becomes trivial and can be eliminated. Therefore, the counting method also supplies a good framework for identifying simple cases where recursive queries with constants can be implemented safely and efficiently by a single fixpoint [AhUl].

5. 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 terminate. Note that the generalized counting method recasts the original query Q into two fixpoint computations: whenever both these computations terminate in a finite number of steps, we will say that *the generalized counting method is safe w.r.t. to the query Q*.

The following property follows immediately from the definitions:

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

We will now assume that our c-predicates are either database predicates or comparison predicates (including equality).

We now give a sufficient condition for the generalized counting method to be safe, which appears to cover most of the situations of practical interest.

Term Length

The length of a term t denoted $|t|$ is defined as follows:

(a) if t is a constant, then $|t|=1$,

(b) if $t=f(t_1, \cdots, t_k)$, then $|t|=|t_1|+\cdots+|t_k|+1$.

This definition allows to determine the length of constant terms. When the terms contain variables, then we can express the length of the term in function of those of the variables. For instance $|x \bullet x| = |x|+|x|+1=2|x|+1$ In general, there is no information on the actual length of x, except that $|x| \geq 1$. Thus $|x \bullet x| \geq 3$.

The length of a set of terms S is the sum of the length of all terms in S. For instance the length of the bound arguments (i.e., $x \bullet y, x_1 \bullet y_1$) in rule r_0 of the *MG* example in Figure 5 is $|x|+|y|+|x_1|+|y_1|+2$.

Arc Length Balance.

Let (R^S, P^T) be an arc in the binding graph with label $[ri, v]$. The length balance associated with this arc is defined as difference between the length of the bound arguments in the head of r_i (i.e., those denoted by S) and the length of the bound arguments of the v-th c-predicate in the body (i.e., the arguments denoted by T). For instance the length balance for the arc labeled $[r_0, 0]$ in the binding graph of Figure 5 is:

$(|x|+|y|+|x_1|+|y_1|+2)-(|y|+|x_1|+|y_1|+1)=|x|+1$.

A lower-bound of the arc length balance can be obtained by replacing the length of the variables by the lower bound of their length if the coefficient is positive, or by the upper bound if the coefficient is negative. For instance, in the previous example, a lower bound of the arc length balance is 2, since the variable x has length one or greater.

Cycle Length Balance.

Given a cycle of the binding graph, the length balance associated to it is defined as the sum of the length balances of its arcs. A lower bound of the cycle length balance can be obtained as the sum of the lower bounds of the arc length balances.

THEOREM 2. *If the the length balance associated with every cycle in the binding graph of a query is positive, then the fixpoint computation of the counting sets converges in a finite number of steps.*

Thus, the examples in Figures 5 and 7 are safe. While Theorem 2 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, there are many situations where more elaborated or completely different techniques must be used.

For instance, if the arc length balance computed over all bound arguments is not positive, one may try to find a subset of the bound arguments for which it is (also a sufficient condition for safety). Often, the cycle length balance depends upon the lengths of variables, which is in turn determined by other predicates (including recursive ones). An interesting technique to deal with some of the more complex situations is given in [UlVg]. For variables that belong to some database predicate, it is often reasonable to assume that their length is one. This additional assumption enables one to infer the safety of the counting method applied to the following example, where Q is a database relation with no function symbols in the second column:

$P(b \bullet b \bullet x)?$

$P(b \bullet b \bullet x)$:-$Q(x,y)$,$P(x \bullet y)$.

$P(b)$.

Finally, there are situations such as those of examples of Figure 4 and 6, where all the solved predicates are database predicates, and the arc balance is null. Therefore, there is no a priori assurance that duplicates cannot occur in the computation of the counting sets. Even for these situations, if the underlying database is known to be acyclic, the generalized counting method remains safe and efficient [SZ1]. When the acyclicity of the underlying database cannot be guaranteed, two solutions are possible. The first is to use methods such as the magic set [BMSU1] and minimagic method [SZ2], that have a built-in check for and elimination of duplicates. The second approach consists of starting with the computation of generalized counting sets while checking for duplicates. If duplicates show up then one will fall back on the standard counting method. This hybrid approach, known as *magic counting* is described in [SZ1].

6. Conclusion

We have presented a new method, named generalized counting, that is very efficient [BR] and appears particularly useful in dealing with recursive rules containing function symbols. The method implements recursive queries by two fixpoint computations. The first propagates the initial bindings into the recursive loop, while the second solves the remaining goals and constructs the desired answer. The method is applicable to arbitrary recursive predicates, including those featuring mutual recursion and non-linear recursion.

The paper also discussed the application of the method to solve nested recursive predicates. A sufficient condition for the finiteness of the fixpoint computations was finally given; although quite simple, this condition seems adequate for many common cases involving recursive predicates with function symbols. It thus appears that the generalized counting method provides a very valuable tool towards compiling pure logic programs with good performance and an a-priori guarantee of termination.

Acknowledgments:
The authors are grateful to François Bancilhon, Ravi Krishnamurthy and Raghu Ramakrishan for many inspiring discussions.

References

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

[AC] Aiello, L. and Cecchi, "Adding a Closure Operator to the Extended Relational Algebra ...", Rome Univ. Technical Report, 1985.

[B] Bancilhon, F., "Naive Evaluation of Recursively defined Relations", Unpublished Manuscript, 1985.

[BGK] Bayer, R., U. Guntzer and W. Kiessling, "On the Evaluation of Recursion in Deductive DB Systems by Efficient Differential Fixpoint Iteration," Technical Report, Technische Univ. Munich, 1985.

[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., Ramakrishan, R., "An amateur's introduction to recursive query processing strategies", *Proc. ACM SIGMOD Int. Conference on Management of Data,* Washington, D.C., May 1986.

[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.

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

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

[L] Lozinškii, E.L., "Inference by generating and structuring of deductive databases", Report 84-11, Dept. of Computer Science, Hebrew

University, Israel.

[MS] McKay, D., Shapiro, S., "Using active connection graphs for reasoning with recursive rules", *Proc. 7th IJCAI, 1981*, pp. 368-374.

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

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

[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., "Implementation of recursive queries for a data language based on pure Horn clauses", unpublished manuscript, 1986.

[SZ3] Saccà, D., Zaniolo, C., "Techniques for Solving Recursive queries in a Logic Based Language," in preparation.

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

[T1] Tarski, A. "A Lattice Theoretical Fixpoint Theorem and its Application," *Pacific Journal of Mathematics No. 5,* pp. 285-309, 1955.

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

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

[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 SIGOD 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.