Theoretical Computer Science 62 (1988) 187-220
North-Holland

THE GENERALIZED COUNTING METHOD FOR RECURSIVE LOGIC QUERIES

Domenico SACCÀ*
Dipartimento di Sistemi, Università della Calabria, Rende, 87036, Italy

Carlo ZANIOLO
Microelectronics and Computer Technology Corporation, Austin, TX 78759, U.S.A.

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 [15, 22]. 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 [27]. However, Prolog's sequential execution model and the spurious nonlogical constructs thus grafted on the language constitute serious drawbacks for database applications since

(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.

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

0304-3975/88/$3.50 © 1988, Elsevier Science Publishers B.V. (North-Holland)

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 ambitious 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 nonrecursive predicates with simple variables, inasmuch as these rules basically correspond to the derived join-select-project views of relational databases [21]. Horn clauses, however, contain two powerful constructs not found in the relational calculus: one is recursion, which, e.g., entails the computation of closures, the other is general unification, which, via the use of function symbols, can be used to support complex and flexible structures (not just flat tuples as in relational databases). The efficient implementation of these two powerful constructs poses some interesting problems [5, 9, 10, 11, 12, 14, 18, 22, 25, 26, 28]. For instance, the technique of using the query constants to search the database efficiently (pushing selection) is frequently inapplicable to recursive predicates [1]. 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 [23, 29].

This paper studies the problem of implementing safely and efficiently recursive Horn clauses in the presence of query constants. For this purpose, it introduces a powerful technique, called the *Generalized Counting Method*, which, in many cases, is more effective in dealing with recursive predicates with function symbols than those previously known [5, 6, 11, 17, 22].

r_0: $\mathrm{SG}(x, y) \text{ :- } P(x, x_1), \mathrm{SG}(x_1, y_1), P(y, y_1).$
r_1: $\mathrm{SG}(x, x) \text{ :- } H(x).$

Fig. 1. The same-generation example.

2. Fixpoint evaluation of recursive queries

Take the recursive rule of Fig. 1, 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

$G1$: $\mathrm{SG}(c, y)?$

defines all persons that are of the same generation. The answer to this query can

[1] A predicate that only unifies with facts will be called a *database predicate*. A *database relation* can be thought of as a set of facts with the same predicate symbol and same number of arguments.

be computed as the least fixpoint of the following function over relations:

$$f(\mathbf{SG}) = \pi_{1,1}\boldsymbol{H} \cup \pi_{1,5}((\boldsymbol{P} \bowtie_{2=1} \mathbf{SG}) \bowtie_{4=2} \boldsymbol{P}). \quad (1)$$

Our function f is defined by a relational algebra expression having as operands the constants $\boldsymbol{H}$ and $\boldsymbol{P}$ and the variable **SG**. ($\boldsymbol{H}$ and $\boldsymbol{P}$ denote the database relations with respective predicate symbols H and P and respective arities 1 and 2—whereas **SG** is an unknown relation with arity 2.) Therefore, the computation of the least fixpoint can proceed by setting the initial value of **SG** to the empty set and computing $f(\mathbf{SG})$. Then $f(\mathbf{SG})$ becomes the new value for **SG** and this iterative step is repeated until no more tuples can be added to **SG**, which then yields 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 [20]. The fixpoint computation approach, refined with the differential techniques, such as those described in [3, 4], 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, it uses only those database facts that were found 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 previously considered is underscored by the safety problem that may arise for logic programs such as the same-generation example without the $H(x)$ goal in the rule r_1 of Fig. 1. In this case, although the answer to the query SG(john, y)? remains the same according to the fixpoint based semantics given in [24], a fixpoint computation is no longer a viable approach for constructing it since in (1) the database relation $\boldsymbol{H}$ is to be replaced by the whole Herbrand universe. This is, in general, large or even infinite when function symbols or comparison predicates are involved. In reality, strategies based on binding propagation (such as the one of Prolog) avoid this safety problem since they only invoke the rule r_1 with the first argument bound. (We note that Prolog may fail if the ordering of predicates in the rules is not appropriate for its resolution strategy.) We propose a method, called the *Generalized Counting Method*, which recasts the query into a

pair of fixpoint computations such that the first computation implements the top-down propagation of bound values and the second one is a modified and safe version of the original bottom-up fixpoint computation. More precisely, the basic approach of the Generalized Counting Method 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 (independently from the ordering of rules and goals).

The Generalized Counting Method also supports the implementation of queries on logic programs with function symbols, such as the merge example of Fig. 2.

r_0: $\mathrm{MG}(x \cdot y, x_1 \cdot y_1, x \cdot w) \text{ :- } \mathrm{MG}(y, x_1 \cdot y_1, w),\ x \geq x_1.$
r_1: $\mathrm{MG}(x \cdot y, x_1 \cdot y_1, x_1 \cdot w) \text{ :- } \mathrm{MG}(x \cdot y, y_1, w),\ x < x_1.$
r_2: $\mathrm{MG}(\mathrm{nil}, x, x).$
r_3: $\mathrm{MG}(x, \mathrm{nil}, x).$

Fig. 2. Merging two sorted lists.

The problem of supporting *nonrecursive* Horn clauses with function symbols was studied in [28]. Since predicates have structured arguments (for instance, the first argument in the head of r_1 in Fig. 2 has x and y as subarguments), an Extended Relational Algebra (ERA) was proposed in [28] 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). Nonrecursive Horn clauses can be implemented as ERA expressions [28, 29]. 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.

In the case of recursive Horn clauses with function symbols, the safety issue on the applicability of the fixpoint approach becomes crucial since the Herbrand universe is infinite. Thus every single step of the fixpoint computation may entail to consider relational expressions having infinite relations as operands. In addition, the safety issue concerned with the termination of the fixpoint computation is to be considered. For instance, the relations representing all possible sets of values for x and x_1 in our rules of Fig. 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 nonterminating one. In reality, safety problems are avoided because a procedure, such as that of

Fig. 2, is only invoked as a goal with certain arguments bound, in order to derive the unbound ones. Typically, for instance, the first two arguments are given to derive the third one.

In conclusion, an effective usage of the binding information available during the top-down phase is vital for performance reasons and to avoid safety problems. The Generalized Counting Method is able to deal with these problems. In particular, this method 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 Fig. 2 cannot be handled with the methods proposed in [22] that do not allow for function symbols on the right side of rules.

3. Binding-passing property

In a logic program LP, a predicate P (with symbol p) is said to *imply* a predicate Q (with symbol q), written $P \rightarrow Q$ if there is a rule in LP with predicate q as the head predicate symbol and predicate symbol p in the body, or there 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 nonrecursive 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 shall call *recursive strong components*, with their graph representation in mind. All predicates in the same recursive strong component must be solved together—they cannot be solved one at a time.

For the LP of Fig. 1, SG is the recursive predicate (a singleton recursive strong component), and H and P are database predicates. However, in the discussion which follows, H and P could be any predicates that can be solved independently of SG; thus they could be derived predicates—even recursive ones—as long as they are not mutually recursive with SG. Finally, it should be clear that here "john" is used as a placeholder for any constant; thus the method proposed here can be used to support any goal with the same binding pattern.

Formally, therefore, we shall study the problem of implementing a query Q that can be modelled as triplet $\langle G, \mathrm{LP}, D \rangle$, where

- LP is a set of Horn clauses, with head predicates all belonging to one recursive strong component, say C;
- G is the goal, consisting of a predicate in C with some bound arguments;
- D is a set of clauses (rules and facts) defining the remaining predicates in the bodies of the LP-rules, which are either nonrecursive or belong to recursive strong components other than C.

The predicates in C will be called the *constructed predicates* (c-predicates for short) and those in D the *datum predicats*. For instance, if our goal is $G2$: SG(john,

x)? on the LP of Fig. 1, then SG is our c-predicate (a singleton recursive strong component) 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 more general predicates, such as comparison predicates and recursive predicates not in the same recursive strong component as the head predicates. Take, for instance, the LP of Fig. 2, with goal $\mathrm{MG}(L_1, L_2, y)$? where L_1 and L_2 denote arbitrary given lists. Here MG is our c-predicate and the comparison predicates $\geqslant$ and $<$ are our datums. The $<$-predicate could, for instance, stand for a database predicate (e.g., if there if a finite set of characters and their lexicographical order is explicitly stored: $a < b$, $b < c, \ldots$) 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 Fig. 3.

$$r_0: \quad x < s(x).$$
$$r_1: \quad x < s(y) :- x < y.$$

Fig. 3. The "less-than" relationship for integers represented by 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 strong component as R; it will be called an *exit rule*, otherwise.

For notational convenience, we shall always index the recusive rules starting from zero and ending with $m-1$, $r_0, \ldots, r_{m-1}$, where m denotes the number of recursive rules under consideration. For instance, in Fig. 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. We assume that our only datums are database and comparison predicates (in Section 4.4, we shall discuss the general case). Then the binding propagation in a rule r_i can be defined as follows. Say that B is a set of variables of r_i (they can be thought of as bound variables). 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*) (a) *Database predicates*: if some variable in a database predicate is in B^+, then all the other variables occurring in that predicate are in B^+.

(b) *Comparison predicates*: if we have an equality, such as $x =$ expression or expression $= x$, and all the variables in expression 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 we shall say that the predicate P is *solved* by B.

Let S be a set of indices denoting *bound* arguments in the head predicate R of r_i, i.e., the index j in S means that the jth argument of R is bound. Let B_S be the set of all variables, appearing in the S-arguments of R.[2] Moreover, let T be the set of indices denoting the arguments bound by B_S in a c-predicate occurrence P in the body of r_i (T may be empty). Then we shall say that r_i *maps the set of bound S-arguments of its head into the set of bound T-arguments of P.*

Say that

(a) V_U denotes the set of all variables appearing in datum predicates of r_i that are not solved by B_S, or in unbound arguments of the head of r_i (i.e., those not in S);

(b) V_C denotes the set of variables appearing in arguments of c-predicates in the body of r_i that are not bound by B_S.

All variables in $\mathrm{UR}_S = B_S^+ \cap (V_U \cup V_C)$ are called the *unreduced* variables in r_i for S and play a crucial role in the Generalized Counting Method, as will be shown later in the paper.

Say, for instance, that the first argument of SG in the head of the rule r_0 in Fig. 1 is bound, thus $S = \{1\}$. Then $B_1 = \{x\}$ and x_1 is bound by $\{x\}$ via the database predicate P, so $B_1^+ = \{x, x_1\}$. Therefore, the bindings propagate from the argument 1 of SG (in the head) to the argument 1 of SG (in the body). $P(x, x_1)$ is solved by B_1 in r_0, whereas $P(y, y_1)$ and $\mathrm{SG}(x_1, y_1)$ are not. Moreover, the set of unreduced variables in the rule r_0 of Fig. 1 for $S = \{1\}$ is empty. Let us now consider the rule r_0 of the example in Fig. 2. If the bound arguments of MG in the head are denoted by $S = 1, 2$, then $B_{1,2} = B_{1,2}^+ = \{x, y, x_1, y_1\}$. The predicate $x \geqslant x_1$ is solved by $B_{1,2}$ whereas the predicate MG in the body is not. We have that $V_U = \{x, w\}$ and $V_C = \{w\}$, so $\mathrm{UR}_{1,2} = B_{1,2}^+ \cap (V_U \cup V_C) = \{x\}$.

To see an example of an unreduced variable appearing in unsolved datum predicates, consider the rule r_0 of the logic program in Fig. 4, where A and B are database predicates. If the bound argument of R is $S = \{1\}$, then $B_1 = \{x\}$ and $B_1^+ = \{x, x_1, x_2, x_3\}$ and the predicates $y \geqslant y_1$ and $x_1 > y$ are not solved by B_1. Moreover, $V_U = \{y, x_1, y_1\}$ and $V_C = \{x_2, y_1\}$ so $\mathrm{UR}_1 = \{x_1, x_2\}$.

r_0: $R(x, y)$:- $A(x, x_1, x_2, x_3)$, $R(x_1, x_2 \cdot y_1)$, $y \geqslant y_1, x_1 < y$.
r_1: $R(x, y)$:- $B(x, y)$.

Fig. 4. A logic program with unreduced variables.

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 labelled by the pair $[r_i, \nu]$, where r_i is the index to a recursive rule, and ν is a zero-base index to c-predicate occurrences in the body of this rule, i.e., 0 is the

[2] By S-arguments we mean the arguments denoted by the set of indices S. Besides, we often represent an index set without parentheses; thus 1 stands for $\{1\}$ and 1, 2 stands for $\{1, 2\}$.

index to the 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 $Q = \langle G(\boldsymbol{x}), \mathrm{LP}, D\rangle$, where $\boldsymbol{x}$ is the list of arguments in the query goal, is constructed as follows:

(i) If S denotes the nonempty set of bound arguments in the query goal $G(\boldsymbol{x})$, 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 with head predicate symbol R that maps the bound arguments of the head predicate into the bound arguments T of the νth c-predicate occurrence and this has symbol P, then P^T is also a node of M_Q, and there is an arc labelled $[r_i, \nu]$ from R^S to P^T.

Figure 5(a) shows the binding graph for a query on the rules of Fig. 1, that binds both arguments. (We refer to this as a query $\mathrm{SG}^{1,2}$ and we shall use the same

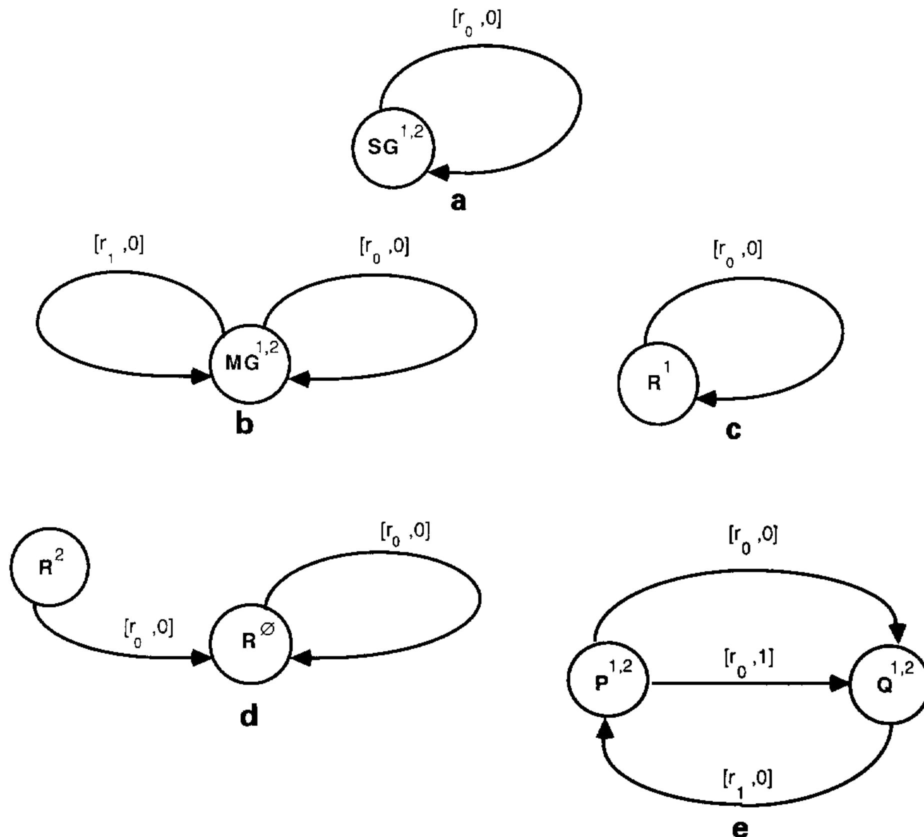


Fig. 5. Binding graphs.

notation for the other queries.) Figure 5(b) shows the graph for a query $MG^{1,2}$ on the rules of Fig. 2. Finally, Figs. 5(c) and 5(d) show the binding graphs for two queries R^1 and R^2 respectively, both on the rules of Fig. 4. Figure 5(e) shows a binding graph for a nonlinear query (i.e., a query having a rule with more than one c-predicate in the body). The query is $P^{1,2}$ and the logic program is shown in Fig. 6.

We can now enunciate our key property.

Binding-passing property: A query Q will be said to have the *binding-passing property* if, for each node R^S of its binding graph, S is not empty.

r_0: $P(x, y) :- B_1(x, x_1, x_3), Q(x_1, y), B_2(x, x_4, x_2), Q(x_2, y), B_3(y, z).$
r_1: $Q(x, y) :- B_4(x, w, z), P(z, y).$
r_2: $P(x, y) :- B_5(x, y).$

Fig. 6. Nonlinear logic program.

Thus the queries whose binding graphs are shown in Figs. 5(a), 5(b), 5(c) and 5(e) respectively have the binding-passing property, whereas the query whose binding graph is in Fig. 5(d) does not.

The binding-passing property guarantees that bindings can be passed down to any level of recursion. Our binding graph is similar to the rule/goal graph described in [22] and is an extension of the binding graph presented in [17]. 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 also be passed through partially bound arguments of c-predicates, but this is left as a topic for future research.

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 3.1. *Let* $Q = \langle G, \mathrm{LP}, D\rangle$ *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 *and* G.

Proof. Let M_Q be the binding graph of Q. We show that the binding graph can be actually constructed in $O(s+g)$ time, where g is the size of the goal G and s is the size of the logic program LP. We start from the query goal G. Obviously, the source node is determined in $O(g)$ time. The other nodes and all arcs are determined as follows. Let R^S be a node in M_Q that has already been determined. Let r_i be a recursive rule such that the predicate symbol of its head is R and let t be the size of the rule. We determine the set B_S of all variables in the arguments S of the head predicate, while reading the rule, in $O(t)$ time. With a little more effort, but yet in

$O(t)$ time, we construct a bipartite undirected graph whose nodes are all predicates in the rule (including the head predicate) and all variables in r_i. There are arcs from each of the above predicates to all variables appearing in it. Besides, if the predicate is an equality predicate $x =$ expression or expression $= x$, we mark every arc from the predicate to the variable x (obviously, there can be at most two marked arcs leaving the equality predicate). Then, we mark all variables in B_S. For each marked variable, we remove all arcs leaving it. Moreover, for each removed arc, if the target node is a database predicate, we mark all variables that are the source of other arcs entering the predicate. If the target node of the removed arc is an equality predicate with only one remaining incoming arc and if this arc is a marked one, we also mark the source (variable) node of the marked arc. We continue the above steps until the arcs leaving all marked (variable) nodes are removed. It is easy to see that the set B_S^+ of variables bound by B_S consists of all marked variables and thus it is constructed in $O(t)$ time. At this point, all arcs leaving R^S in M_Q with label r_i and the corresponding target nodes can be easily constructed by just determining the bound arguments in every c-predicate in the body of r_i, i.e., those arguments that do not contain unmarked variables. Therefore, the graph M_Q can be constructed in $O(g + 2^{\hat{k}} \times s)$ time since the total sum of the size of the rules is equal to the size s of LP and the same rule can be used to generate arcs for at most $2^{\hat{k}}$ different index sets S, where $\hat{k}$ is the maximum predicate arity. Since $\hat{k}$ is bound by hypothesis, the graph is constructed in time linear in the size of the logic program and the query goal. Obviously, testing whether every node R^S has an empty index set S can be done in $O(s + g)$ time while constructing the graph, thus the binding-passing property can be tested in linear time as well. □

We now introduce a characterization of the nodes in the binding graph that will be used later in the description of the Generalized Counting Method. Let Q be a query and M_Q be its binding graph. A node R^S in M_Q is

(a) *strongly reduced* if, for every arc $(A^{\hat{S}}, P^{\hat{T}})$ (say with label $[r_i, \nu]$) in M_Q such that the node R^S is reachable from $P^{\hat{T}}$, no database predicate in the body of r_i is solved by $B_{\hat{S}}$,[3] thus the binding is directly propagated without using database predicates;

(b) *reduced* if, for every arc $(Q^{\hat{S}}, R^S)$ (say with label $[r_i, \nu]$), either

  (i) the rule r_i has only one c-predicate in its body and there are no unreduced variables in r_i for S, or

  (ii) $Q^{\hat{S}}$ is strongly reduced and no database predicate in the body of r_i is solved by $B_{\hat{S}}$. Also in such cases, the binding is directly propagated without using database predicates.

Obviously, if a node is strongly reduced, it is also reduced. We point out that strongly reduced nodes arise in many logic programs with function symbols (see, for instance, the list merge example of Fig. 2).

[3] Recall that $B_{\hat{S}}$ is the set of all variables appearing in the $\hat{S}$-arguments of the head of r_i.

A binding graph is *reduced* if all its nodes are reduced.

Consider the query SG^1 on the logic program of Fig. 1, whose binding graph is similar to the one shown in Fig. 5(a). The node of this graph is reduced because the rule r_i has no unreduced variables for $S=\{1\}$; so the binding graph is reduced. Consider now the node $MG^{1,2}$ in the binding graph of Fig. 5(b). This node is strongly reduced since the rules labelling its incoming arcs have no solved database predicates. So this binding graph is reduced. The node R^1 in the binding graph of Fig. 5(d) is not reduced since the rule r_0 in Fig. 4 has two unreduced variables for $S=\{1\}$. Therefore, this binding graph is not reduced. Finally, consider the binding graph of Fig. 5(e). The node $P^{1,2}$ is not strongly reduced, so the node $Q^{1,2}$ (appearing in a nonlinear rule) is not reduced. Hence, the graph is not reduced.

Proposition 3.2. *Let $Q=\langle G, \mathrm{LP}, D\rangle$ be a query such that there is a bound on the arity of the predicates in* LP. *Then testing whether the binding graph of Q is reduced can be done in time $O(s+m^2)$, where s is the size of* LP *and m is the number of recursive rules in* LP.

Proof. We first of all determine strongly reduced nodes. For every arc $(Q^{\hat{S}}, P^{\hat{T}})$ (say with label $[r_i, \nu]$), we construct $B^+_{\hat{S}}$ in $O(t)$ time, where t is the size of the rule r_i (see the algorithm sketched in the proof of Proposition 3.1). Then we test whether there is some database predicate in r_i that is solved by $B_{\hat{S}}$; if not, we mark the arc. Obviously, testing and marking the arc can be done in $O(t)$ time. So the complexity of marking all arcs is $O(2^{\hat{k}}\times s)$, where $\hat{k}$ is the maximum predicate arity in LP. We now reverse the direction of all arcs in the binding graph. A node R^S is strongly reduced if all arcs reachable from it are marked. Using a transitive closure algorithm, it is easy to see that condition (a) can be tested in time $O(2^{\hat{k}}\times m\times 2^{\hat{k}}\times m)$, because the number of both nodes and arcs is bound by $2^{\hat{k}}\times m$. Hence, since $\hat{k}$ is bound by hypothesis, the strongly reduced nodes are determined in time $O(s+m^2)$. In order to check whether a node R^S is reduced, we only need to consider every arc entering R^S, say $(Q^{\hat{S}}, R^S)$ with label $[r_i, \nu]$ and to construct the set of unreduced variables in r_i for T. Again, this step can be easily done in $O(t)$ time, where t is the size of the rule r_i. On the other hand, it is already known whether or not $Q^{\hat{S}}$ is strongly reduced and the number of c-predicates in the body of r_i can be computed in $O(t)$ time. Therefore, the condition for R^S to be reduced can be tested in $O(t)$ time. It follows that all reduced nodes can be determined in time $O(2^{\hat{k}}\times s)$ and, then, in $O(s)$ time. Hence, testing whether the binding graph is reduced can be done in $O(s+m^2)$. □

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 [17] 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 [5].

The generalized counting method recasts a query into a pair of fixpoint computations that are, in general, more efficient than computing the query via a single fixpoint computation or using other techniques for its resolution (see [7, 13] for a description of the efficient behavior of the method). Besides, every step of the two fixpoint computations is safe (i.e., it can be carried out in a finite amount of time) in a larger number of cases with respect to the single fixpoint computation (see Section 5.1). Finally, there are sufficient conditions for the termination of the two fixpoint computations that cover a large number of cases.

While the pair of fixpoint computations could be expressed directly in terms of relational algebra [17], 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 $Q = \langle G, \mathrm{LP}, D\rangle$ into an equivalent query $\bar{Q} = \langle G, \overline{\mathrm{LP}}, D\rangle$ that can be, in general, computed safely and efficiently using the fixpoint approach described in Section 2. In $\overline{\mathrm{LP}}$ 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 answer.

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 body of the rule.

Exit counting rules

The first counting rule is generated by the source node in M_Q, say $\hat{G}^S$, which corresponds to the query goal. Say that the query goal has $n = |S| \geq 1$ bound arguments with respective values $a_1, \ldots, a_n$.[4] Then we introduce the following clause

[4] Here and elsewhere, arguments as well as variables are listed in the order they appear in the original rule.

for the counting set:

$$\text{cnt}.\hat{G}^S(0, 0, 0, a_1, \ldots, a_n).$$

Recursive counting rules

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

$$\text{cnt}.P^T(j+1, m \times k + i, p \times h + \nu, y_1, \ldots, y_l)$$
$$:\text{-}\ \text{cnt}.R^S(j, k, h, x_1, \ldots, x_n), Q_1, \ldots, Q_q$$

where

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

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

(iii) $Q_1, \ldots, Q_q$ are all datum predicates of r_i that are solved by the set of all variables in the arguments S of the head predicate.

(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 indices 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 $\text{cnt}.P^T(j', k', h', y_1, \ldots, y_l)$, 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 unsolved datum predicates and all c-predicates in the body but the one under consideration by exchanging this c-predicate with that in the head, and by replacing the unbound arguments of these two c-predicates with the three indices. 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. Figures 7(a), 7(b), 7(c) and 7(d) show the counting rules for the queries $SG^{1,2}$, $MG^{1,2}$, R^1 and P^1 on the logic programs of Figs. 1, 2, 4 and 6 respectively. The figures also contain supplementary counting rules that will be explained next.

In the top-down generation of the counting sets we often generate additional values that are needed in the successive bottom-up computation. For instance, in the merge example of Fig. 7, we generate the values of x in rule r_0 and those of x_1 in r_1 that must be saved for later use in the bottom-up phase (note that such variables are unreduced).

Supplementary counting variables

Consider a node R^S in M_Q. In M_Q, there is a bundle of arcs leaving R_S labelled with the same rule r_i, one arc for each c-predicate in the body of r_i. We may need

Counting rules:

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

Supplementary counting rules: None

(a) Rules for the query SG(a, b)?

Counting rules;

$\text{cnt.MG}^{1,2}(0, 0, 0, L_1, L_2).$
$\text{cnt.MG}^{1,2}(j+1, 2\times k+0, 1\times h+0, y, x_1\cdot y_1)$:- $\text{cnt.MG}^{1,2}(j, k, h, x\cdot y, x_1\cdot y_1),\ x \geqslant x_1.$
$\text{cnt.MG}^{1,2}(j+1, 2\times k+1, 1\times h+0, x\cdot y, y_1)$:- $\text{cnt.MG}^{1,2}(j, k, h, x\cdot y, x_1\cdot y_1), x < x_1.$

Supplementary counting rules:

$\text{spcnt.MG}^{1,2}.r_0(j+1, 2\times k+0, 1\times h+0, x)$:- $\text{cnt.MG}^{1,2}(j, k, h, x\cdot y, x_1\cdot y_1),\ x \geqslant x_1.$
$\text{spcnt.MG}^{1,2}.r_1(j+1, 2\times k+0, 1\times h+0, x_1)$:- $\text{cnt.MG}^{1,2}(j, k, h, x\cdot y, x_1\cdot y_1), x < x_1.$

(b) Rules for the query MG(L_1, L_2, w)?

Counting rules:

$\text{cnt.}R^1(0, 0, 0, a).$
$\text{cnt.}R^1(j+1, k, h, x_1)$:- $\text{cnt.}R^1(j, k, h, x), A(x, x_1, x_2, x_3).$

Supplementary counting rules:

$\text{spcnt.}R^1.r_0(j, k, h, x, x_1, x_2)$:- $\text{cnt.}R^1(j, k, h, x), A(x, x_1, x_2, x_3).$

(c) Rules for the query $R(a, y)$?

Counting rules:

$\text{cnt.}P^1(0, 0, 0, a).$
$\text{cnt.}Q^1(j+1, 2\times k, 2\times h, x_1)$:- $\text{cnt.}P^1(j, k, h, x), B_1(x, x_1, x_3), B_2(x, x_4, x_2).$
$\text{cnt.}Q^1(j+1, 2\times k, 2\times h+1, x_2)$:- $\text{cnt.}P^1(j, k, h, x), B_1(x, x_1, x_3), B_2(x, x_4, x_2).$
$\text{cnt.}P^1(j+1, 2\times k+1, 2\times h, z)$:- $\text{cnt.}Q^1(j, k, h, x), B_4(x, w, z).$

Supplementary counting rules:

$\text{spcnt.}P^1.r_0(j, k, h, x, x_1, x_2)$:- $\text{cnt.}P^1(j, k, h, x), B_1(x, x_1, x_3), B_2(x, x_4, x_2).$
$\text{spcnt.}Q^1.r_1(j, k, h, x, z)$:- $\text{cnt.}Q^1(j, k, h, x), B_4(x, w, z).$

(d) Rules for the query $P(a, y)$?

Fig. 7. Counting and supplementary counting rules.

to save the values of some variables in r_i for the subsequent bottom-up phase. The set of such variables, called *supplementary counting variables*, is denoted by V_{sp} and is defined as follows. Let B_S be the set of all variables appearing in the bound arguments of the head of r_i (i.e., those in S), let UR_S be the set of unreduced variables in r_i for S, and B_C be the set of all variables appearing in some bound argument of a c-predicate in the body of r_i that is bound by B_S.

(a) If UR_S is not empty and M_Q is reduced, then $V_{sp} = \text{UR}_S$ (see the example in Fig. 7(b)).
(b) If UR_S is not empty and M_Q is not reduced, then $V_{sp} = \text{UR}_S \cup B_C \cup B_S$ (see the example in Fig. 7(c)).

(c) If UR_S is empty and M_Q is reduced, then V_{sp} is empty (see the example in Fig. 7(a)).

(d) If UR_S is empty and R^S is not reduced, then $V_{\mathrm{sp}} = B_C \cup B_S$ (see the example in Fig. 7(d)).

Supplementary counting rules

We shall add a supplementary counting rule for each bundle of arcs labelled 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 labelling a bundle leaving, say, node R^S, then, using the counting set for R^S, we write

$$\mathrm{spcnt}.r_i.R^S(j, k, h, z_1, \ldots, z_t) \text{ :- } \mathrm{cnt}.R^S(j, k, h, x_1, \ldots, x_n), Q_1, \ldots, Q_q,$$

where

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

(ii) $z_1, \ldots, z_t$ are the supplementary counting variables,

(iii) $Q_1, \ldots, Q_q$ are the datum predicates of r_i that are solved by the set of all variables appearing in the arguments S of the head predicate of r_i.

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 labelled 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 succeeds only if the remainder is zero):

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

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) if the binding graph is reduced; otherwise they are all arguments in the head;

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

(iii) $W_1, \ldots, W_q$ are the datum predicates of r_i that are not solved by the set of all variables appearing in the arguments S of the head predicate;

(iv) $\hat{P}_0, \ldots, \hat{P}_{p-1}$ are the modified c-predicate occurrences in the body of r_i, constructed as follows: say that there is an arc labelled $[r_i, \nu]$ from R^S to P^T, and $x_1, \ldots, x_l$ are the unbound arguments in the νth c-predicate of r_i (i.e., those not in

T) if the binding graph is reduced, otherwise they are all the arguments in the predicate; then

$$\hat{P}_\nu = P^T(j, k, h+\nu, x_1, \ldots, x_l);$$

(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 labelling 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 datum predicates, remove the bound arguments of all c-predicates (including the one in the head) if the binding graph is reduced, introduce the three indexes (after suitable indexing operations) and, finally, add the supplementary counting predicate, if any.

Figures 8(a), 8(b), 8(c) and 8(d) show the modified recursive rules for the usual queries $\mathrm{SG}^{1,2}$, $\mathrm{MG}^{1,2}$, R^1 and P^1. These figures also contain modified exit rules and goal rules that will be explained next.

Modified exit rules

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

$$R^S(j, k, h, u_1, \ldots, u_n) \text{ :- } \mathrm{cnt}.R^S(j, k, h, x_1, \ldots, x_l), W_1, \ldots, W_q,$$

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) if the binding graph is reduced; otherwise they are all the arguments of the head predicate;

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

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

Thus the exit rules are generated by introducing the indices, removing the bound arguments in the head if the binding graph is reduced, and then adding the counting set to the body of the rule.

Goal rule

In order to unify the initial query goal with the corresponding modified c-predicate, a new rule, called the goal rule, is added to $\overline{\mathrm{LP}}$. If $\hat{G}(u_1, \ldots, u_p)$ is the original query goal with bound arguments denoted by S, then the goal rule is

$$\hat{G}(u_1, \ldots, u_p) \text{ :- } \hat{G}^S(0, 0, 0, x_1, \ldots, x_q),$$

where

(1) $q = p$ and $x_1 = u_1, \ldots, x_p = u_p$ if the binding graph is not reduced, or

(2) $x_1, \ldots, x_q$ are the unbound arguments in $\hat{G}^S$ if the binding graph is reduced (recall that $\hat{G}^S$ is the source node of the binding graph).

Modified recursive rule:

$SG^{1,2}(j-1, k, h) :\text{-}\ SG^{1,2}(j, k, h).$

Modified exit rule:

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

Goal rule:

$SG(a, b) :\text{-}\ SG^{1,2}(0, 0, 0).$

(a) Rules for the query $SG(a, b)$?

Modified recursive rules:

$MG^{1,2}(j-1, \frac{1}{2}k, h, x \cdot w) :\text{-}\ \text{spcnt.}MG^{1,2}.r_0(j-1, \frac{1}{2}k, h, x), MG^{1,2}(j, k, h, w).$
$MG^{1,2}(j-1, \frac{1}{2}(k-1), h, x_1 \cdot w) :\text{-}\ \text{spcnt.}MG^{1,2}.r_1(j-1, \frac{1}{2}(k-1), h, x_1), MG^{1,2}(j, k, h, w).$

Modified exit rules:

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

Goal rule:

$MG(L_1, L_2, w) :\text{-}\ MG^{1,2}(0, 0, 0, w).$

(b) Rules for the query $MG(L_1, L_2, w)$?

Modified recursive rule:

$R^1(j-1, k, h, x, y) :\text{-}\ \text{spcnt.}R^1.r_0(j-1, k, h, x, x_1, x_2), R^1(j, k, h, x_1, x_2 \cdot y_1),\ y \geq y_1, x_1 < y.$

Modified exit rule:

$R^1(j, k, h, x, y) :\text{-}\ \text{cnt.}R^1(j, k, h, x), B(x, y).$

Goal rule:

$R(a, y) :\text{-}\ R^1(0, 0, 0, a, y).$

(c) Rules for the query $R(a, y)$?

Modified recursive rules:

$P^1(j-1, \frac{1}{2}k, \frac{1}{2}h, x, y) :\text{-}\ \text{spcnt.}P^1.r_0(j-1, \frac{1}{2}k, \frac{1}{2}h, x, x_1, x_2), Q^1(j, k, h, x_1, y), Q^1(j, k, h+1, x_2, y), B_3(y, z).$

$Q^1(j-1, \frac{1}{2}(k-1), h, x, y) :\text{-}\ \text{spcnt.}Q^1.r_1(j-1, \frac{1}{2}(k-1), h, x, z), P^1(j, k, h, z, y).$

Modified exit rule:

$P^1(j, k, h, x, y) :\text{-}\ \text{cnt.}P^1(j, k, h, x), B_5(x, y).$

Goal rule:

$P(a, y) :\text{-}\ P^1(0, 0, 0, a, y).$

(d) Rules for the query $P(a, y)$?

Fig. 8. Modified and goal rules.

It is easy to see that the fixpoint computation of the modified rules should be stopped once the indices are returned to zero.

4.3. *Properties of the generalized counting method*

For the purpose of proving the correctness of the generalized counting method we can assume that all the datum predicts in D are defined by facts only. Database predicates are defined by a finite number of facts in D. We can also assume [29] that comparison predicates are defined by a (infinite) number of facts in D (possibly with complex arithmetic terms). Finally, all the rules used in defining derived predicates in D can be replaced by the set of facts that can be inferred using said rules, without changing the answer to the given query Q. In this framework, we can focus on the fixpoint-based semantics of the original program vis-à-vis that of the program produced by the Generalized Counting Method.

Let $Q = \langle G, \mathrm{LP}, D\rangle$ be a query as before and let H be the Herbrand universe of $\mathrm{LP} \cup D$ (in general, H is infinite and contains complex terms). Consider a fact $R(\boldsymbol{a})$ where $\boldsymbol{a}$ is a list of bound arguments. A *derivation tree* for $R(\boldsymbol{a})$ is defined as follows. Each node of the tree is a fact whose predicate symbol appears in $\mathrm{LP} \cup D$ and the root is $R(\boldsymbol{a})$. Every non-leaf node (say N)is labelled by a rule (say r_i) in LP whose head predicate symbol is equal to that of the node. Besides, N has as many children as predicates in the body of r_i, each child being a fact with the symbol of the corresponding predicate. Every leaf node either is a fact in D (*datum node*) or is labelled by an exit rule r_i with empty body. Every node N labelled by a rule r_i is *solved in* r_i, thus there is an assignment of all variables in r_i to the elements of H such that, after replacing the variables with the assigned terms, the head predicate equals N and every predicate in the body (if any) is equal to the corresponding child of N. Such a variable assignment is called a *solving assignment* for N.

We say that $R(\boldsymbol{a})$ is inferred from $\mathrm{LP} \cup D$ if there is a (finite) derivation tree for $R(\boldsymbol{a})$. It also turns out that every fact in a derivation tree is inferred from $\mathrm{LP} \cup D$ as well.

Let $Q = \langle G(\boldsymbol{x}), \mathrm{LP}, D\rangle$ be a query, where $\boldsymbol{x}$ is a list of arguments. The set of *answers* of Q is the set of all facts $G(\boldsymbol{a})$ such that $\boldsymbol{a}$ is a list of bound arguments, $G(\boldsymbol{a})$ is inferred from $\mathrm{LP} \cup D$, and $G(\boldsymbol{x})$ unifies with $G(\boldsymbol{a})$.

Two queries are *equivalent* if they have the same set of answers. Let $Q = \langle G, \mathrm{LP}, D\rangle$ be a query that has the binding-passing property and let $\bar{Q} = \langle G, \overline{\mathrm{LP}}, D\rangle$ be the modified query produced by the generalized counting method. We now show that Q and $\bar{Q}$ are equivalent. To this end, we need to introduce some notation and four technical lemmata.

Let $\boldsymbol{a}$ be a list of arguments[5] and let S be a set of indices denoting some of these arguments. Then $\boldsymbol{a}^S$ denotes the list of all arguments of $\boldsymbol{a}$ indexed by S and $\boldsymbol{a}^{S^-}$

[5] From now on, we use bold italic letters to denote lists of arguments. Moreover, we use the first letters of the alphabet to denote bound arguments.

denotes all the others. For instance, if $\boldsymbol{a} = \langle a, b, c, d, e\rangle$ and $S = \{1, 3, 4\}$, then $\boldsymbol{a}^S = \langle a, c, d\rangle$ and $\boldsymbol{a}^{S^-} = \langle b, e\rangle$.

A fact whose symbol is that of a c-predicate or a modified c-predicate is called a *c-fact*. A fact whose symbol is that of a counting predicate or a supplementary counting predicate is called *counting fact* and *supplementary counting fact* respectively.

Lemma 4.1. *Let* $Q = \langle G, \mathrm{LP}, D\rangle$ *be a query that has the binding-passing property and let* $\bar{Q} = \langle G, \overline{\mathrm{LP}}, D\rangle$ *be the modified query produced by the Generalized Counting Method. If* $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{b})$ *is a c-fact inferred from* $\overline{\mathrm{LP}} \cup D$, *then any derivation tree of* $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{b})$ *contains a counting fact* $\mathrm{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$.

Proof. We carry out the proof by induction on the number of c-facts appearing in the derivation tree of $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{b})$.

Basis of the induction: only one c-fact in the tree. This c-fact coincides with the root, i.e., $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{b})$. Since only a modified rule in $\overline{\mathrm{LP}}$ can unify with a modified c-predicate and since modified recursive rules have c-predicates in their body, the root is labelled by a modified exit rule, say

$$R^S(j, k, h, \boldsymbol{u}) \text{ :- } \mathrm{cnt}.R^S(j, k, h, \boldsymbol{x}),\ W_1, \ldots, W_q.$$

Hence, by the definition of a derivation tree, one of the children of $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{b})$ is a counting fact $\mathrm{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$.

Induction step: The lemma holds whenever a derivation tree has less than s c-facts, where $s > 1$. We now show that it also holds when the derivation tree has s c-facts. Let us denote this tree by DT. Since $s > 1$, the root $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{b})$ of DT is labelled by a modified recursive rule, say $\hat{r}_i$,

$$R^S(j, k, h, \boldsymbol{u}) \text{ :- } \mathrm{spcnt}.R^S.r_i(j, k, h, z),\ \hat{P}_0, \ldots, \hat{P}_{p-1},\ W_1, \ldots, W_q.$$

We recall that the supplementary counting predicate may be missing. Furthermore, the three indices for the c-predicate $\hat{P}_\nu$, $0 \leqslant \nu \leqslant p-1$, are $j+1$, $k \times m + i$ and $h \times p + \nu$. Because of the definition of derivation tree, one of the children of the root is a c-fact corresponding to $\hat{P}_0$, say $P_0^{T_0}(\hat{j}+1, \hat{k} \times m + i, \hat{h} \times p, \boldsymbol{c})$. The subtree rooted at this node is a derivation tree for it and contains less than s facts. Hence, by inductive hypothesis, there exists a counting fact $\mathrm{cnt}.P_0^{T_0}(\hat{j}+1, \hat{k} \times m + i, \hat{h} \times p, \boldsymbol{d})$ in the subtree and, then, in DT.

Consider now the rule r_i in LP associated to $\hat{r}_i$. We have that r_i is

$$R(\boldsymbol{t}) \text{ :- } Q_1, \ldots, Q_r, P_1, \ldots, P_{p-1}, W_1, \ldots, W_q.$$

where $Q_1, \ldots, Q_r$ are the datum predicates solved by the variables in the S-arguments of $R(\boldsymbol{t})$ and P_ν, $0 \leqslant \nu \leqslant p-1$ are the c-predicates corresponding to $\hat{P}_\nu$, $0 \leqslant \nu \leqslant p-1$.

Clearly, since $p \geq 1$, there is an arc from R^S to $P_0^{T_0}$ with label $[r_i, 0]$. The associated counting rule is

$$\text{cnt}.P_0^{T_0}(j+1, m \times k + i, p \times h, \boldsymbol{y}) \ \text{:-} \ \text{cnt}.R^S(j, k, h, \mathbf{x}), Q_1, \ldots, Q_r.$$

Considering the structure of the indices, this rule is the only one that can label the counting fact $\text{cnt}.P_0^{T_0}(\hat{j}+1, \hat{k} \times m + i, \hat{h} \times p, \boldsymbol{d})$. It follows that one of the children of this node is a counting fact with predicate symbol $\text{cnt}.R^S$ and indices $\hat{j}, \hat{k}, \hat{h}$. Hence, a counting fact $\text{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$ is in DT and this concludes the proof. □

Lemma 4.2. *Let $Q = \langle G, \text{LP}, D \rangle$ be a query that has the binding-passing property and whose binding graph M_Q is reduced. Let $\bar{Q} = \langle G, \overline{\text{LP}}, D \rangle$ be the modified query produced by the generalized counting method. Let R^S be a strongly reduced node of M_Q and let* $\text{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$ *be a counting fact inferred from* $\text{LP} \cup D$. *Then no other counting fact* $\text{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{b})$ *with $\boldsymbol{a} \neq \boldsymbol{b}$ is inferred from* $\text{LP} \cup D$.

Proof. We prove the lemma by induction on the depth of the derivation tree for $\text{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$. The basis is a derivation tree where the only node is the root. Obviously, $\hat{j} = \hat{k} = \hat{h} = 0$ and the proof is trivial. Suppose now that the lemma holds for every derivation tree with depth less than s, $s > 0$ (inductive hypothesis). Let DT be a derivation tree for $\text{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$ with depth equal to s. We now proceed by contradiction. Suppose that there is another counting fact $\text{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{b})$ with $\boldsymbol{b} \neq \boldsymbol{a}$ that is inferred from $\text{LP} \cup D$. Let DT′ be any derivation tree for $\text{cnt}.R^S(\hat{j}, \hat{k}, h', \boldsymbol{b})$. Because of the structure of the indices, the depth of DT′ is s and the roots of DT and DT′ are labelled by the same recursive counting rule, say the following rule $\hat{r}_i$:

$$\text{cnt}.R^S(j', k', h', \boldsymbol{x}) \ \text{:-} \ \text{cnt}.P^T(j', k', h', \boldsymbol{y}), Q_1, \ldots, Q_r,$$

where, by the definition of a strongly reduced node, $Q_1, \ldots, Q_r$ (if present) are comparison predicates. Therefore, the children of the root in both DT and DT′ are a counting fact (say $\text{cnt}.P^T(\bar{j}, \bar{k}, \bar{h}, \boldsymbol{c})$ and $\text{cnt}.P^T(\bar{j}, \bar{k}, \bar{h}, \boldsymbol{d})$ respectively) and, possibly, the nodes corresponding to the above comparison predicates. Since all the variables in the comparison predicates are bound by those in the arguments $\boldsymbol{y}$ and every variable in $\boldsymbol{x}$ appears in the body of $\hat{r}_i$ as well, $\boldsymbol{b} \neq \boldsymbol{a}$ implies $\boldsymbol{c} \neq \boldsymbol{d}$. On the other hand, from the fact that R^S is reduced and from the definition of strongly reduced node, it follows that P^T is strongly reduced as well. So, by inductive hypothesis, $\boldsymbol{c} = \boldsymbol{d}$ (contradiction). Therefore, $\boldsymbol{b} = \boldsymbol{a}$ and this concludes the proof. □

Lemma 4.3. *Let $Q = \langle G, \text{LP}, D \rangle$ be a query that has the binding-passing property and whose binding graph is M_Q. Let $\bar{Q} = \langle G, \overline{\text{LP}}, D \rangle$ be the modified query produced by the Generalized Counting Method. Then, for every c-fact $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$ that is inferred from $\overline{\text{LP}} \cup D$, there exists a fact $R(\boldsymbol{b})$ such that $R(\boldsymbol{b})$ is inferred from* $\text{LP} \cup D$ *and*

(a) *$\boldsymbol{b} = \boldsymbol{a}$ if M_Q is not reduced, or*

(b) $\boldsymbol{b}^S = \boldsymbol{c}$ *and* $\boldsymbol{b}^{S^-} = \boldsymbol{a}$, *where* $\text{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{c})$ *is a counting fact inferred from* $\overline{\text{LP}} \cup D$ *if* M_Q *is reduced.*

Proof. We proceed by induction on the number of c-facts in the derivation tree of $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$.

Basis of the induction: only one c-fact in the derivation tree. Hence, the only c-fact is the root, i.e., $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$, and it is labelled by a modified exit rule $\hat{r}_i$ in $\overline{\text{LP}}$ say,

$$R^S(j, k, h, \boldsymbol{u}) \text{ :- } \text{cnt}.R^S(j, k, h, \boldsymbol{x}), W_1, \ldots, W_q.$$

Therefore, one of the children of the root is a counting fact with indices $\hat{j}$, $\hat{k}$ and $\hat{h}$, say $\text{cnt}.R^D(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{c})$. We now modify this tree as follows. We remove the subtree rooted at this counting fact. Besides, we replace the root $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$ with $R(\boldsymbol{b})$, where $\boldsymbol{b} = \boldsymbol{a}$ if M_Q is not reduced or $\boldsymbol{b}^S = \boldsymbol{c}$ and $\boldsymbol{b}^{S} = \boldsymbol{a}$ if M_Q is reduced. On the other hand, the leaf nodes corresponding to $W_1, \ldots, W_q$ remain unchanged. Obviously, such nodes are facts in D. Finally, we label the root of the new tree after the corresponding exit rule r_i in the original program LP. In order to show that the so-obtained tree is actually a derivation tree for $R(\boldsymbol{b})$, we have to prove that there is a solving assignment for $R(\boldsymbol{b})$ in r_i. Consider the solving assignment ϕ for $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$ in $\hat{r}_i$. By construction, the set of all variables appearing in $\hat{r}_i$ but the index variables j, k, h is equal to the set of all variables in r_i. Therefore, the restriction of ϕ to all variables in $\hat{r}_i$ but j, k, h is a solving assignment for $R(\boldsymbol{b})$ in r_i. It follows that the modified tree is a derivation tree for $R(\boldsymbol{b})$, thus $R(\boldsymbol{b})$ is inferred from $\text{LP} \cup D$.

Induction step: the lemma holds for any derivation tree with less than s c-facts, where $s > 1$ (inductive hypothesis). We prove that it also holds for derivation tree with s c-facts. Let us assume that a derivation tree DT for $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$ contains s c-facts. Hence, the root is labelled by a modified recursive rule, say the following rule $\hat{r}_i$

$$R^S(j, k, h, \boldsymbol{u}) \text{ :- } \text{spcnt}.R^S.r_i(j, k, h, z), \hat{P}_0, \ldots \hat{P}_{p-1}, W_1, \ldots, W_q.$$

The original rule in LP is r_i

$$R(\boldsymbol{t}) \text{ :- } Q_1, \ldots, Q_r, P_1, \ldots, P_{p-1}, W_1, \ldots, W_q.$$

We recall that if the list z of supplementary counting variables is empty, then the supplementary counting predicate is missing. Furthermore, the three indices for the c-predicate $\hat{P}_\nu$, $0 \leq \nu \leq p-1$, are $j+1$, $k \times m + i$ and $h \times p + \nu$. We modify DT into a new tree DT′ as follows. The possible subtree rooted at the node corresponding to $\text{spcnt}.R^S.r_i(j, k, h, z)$ is pruned, whereas the leaf nodes corresponding to $W_1, \ldots, W_q$ remain unchanged. The other children of $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$ are the c-facts corresponding to $\hat{P}_0, \ldots, \hat{P}_{p-1}$. For every $\hat{P}_\nu$, $0 \leq \nu \leq p-1$, the subtree rooted at the corresponding c-fact, say $P_\nu^{T_\nu}(\hat{j}+1, \hat{k} \times m + i, \hat{h} \times p + \nu, \boldsymbol{d})$, contains a counting fact $\text{cnt}.P_\nu^{T_\nu}(\hat{j}+1, \hat{k} \times m + i, \hat{h} \times p + \nu, \boldsymbol{e})$ by Lemma 4.1. We replace this subtree by a derivation tree for $P_\nu(\boldsymbol{f})$ from $\text{LP} \cup D$, where $\boldsymbol{f} = \boldsymbol{d}$ if M_Q is not reduced, or $\boldsymbol{f}^T = \boldsymbol{e}$ and $\boldsymbol{f}^T = \boldsymbol{d}$ if

M_Q is reduced. By the inductive hypothesis, this derivation tree exists. To complete DT′, we only need to modify the root of DT and to add leaf nodes corresponding to the datum predicates $Q_1, \ldots, Q_r$. To this end, we distinguish two possible cases:

Case 1: the binding graph M_Q is reduced. The derivation tree DT contains the counting fact cnt.$P_0^{T_0}(\hat{j}+1, \hat{k}\times m+i, \hat{h}\times p, \boldsymbol{e})$. This node is labelled by the recursive counting rule associated to the arc from R^S to $P_0^{T_0}$ with label $[r_i, 0]$; thus,

$$\text{cnt}.P_0^{T_0}(j+1, m\times k+i, p\times h, \boldsymbol{y}) \text{ :- } \text{cnt}.R^S(j, k, h, \boldsymbol{x}), Q_1, \ldots, Q_r.$$

The children of this node are the leaf nodes corresponding to $Q_1, \ldots, Q_r$ and a counting fact with predicate symbol cnt.R^S and indices $\hat{j}$, $\hat{k}$ and $\hat{h}$, say cnt.$R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{c})$. Obviously, this counting fact is inferrred from $\overline{\text{LP}} \cup D$. We replace the root $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$ with $R(\boldsymbol{b})$, where $\boldsymbol{b}^S = \boldsymbol{c}$ and $\boldsymbol{b}^{S^-} = \boldsymbol{a}$. Finally, we add the above leaf nodes corresponding to $Q_1, \ldots, Q_r$ into DT′ as children of the root.

Case 2: the binding graph M_Q is not reduced. Hence, the body of $\hat{r}_i$ contains the predicate spcnt.$R^S.r_i(j, k, h, z)$. Then we consider the child node of $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$, corresponding to the supplementary counting predicate. This node is labelled by the following supplementary counting rule:

$$\text{spcnt}.r_i.R^S(j, k, h, z) \text{ :- } \text{cnt}.R^S(j, k, h, \boldsymbol{x}), Q_1, \ldots, Q_r.$$

We proceed as for Case 1; so we single out a counting fact cnt.$R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{c})$ and we add the leaf nodes corresponding to $Q_1, \ldots, Q_r$ into DT′ as children of the root. Finally, we replace the root $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$ with $R(\boldsymbol{b})$, where $\boldsymbol{b} = \boldsymbol{a}$.

In both cases, we label the new root by the recursive rule r_i in LP. Since all leaf children of the root are in D by construction and since all other children are root of derivation (sub)trees, we only need to prove that the root $R(\boldsymbol{b})$ is solved in r_i. Thus, we have to construct a solving assignment ϕ for the variables in r_i. Let $\hat{\phi}$ be the solving assignment for $R^S(\hat{j}, \hat{h}, \hat{k}, \boldsymbol{a})$ in $\hat{r}_i$. Let $V_{\hat{r}_i}$ be the set of all variables in $\hat{r}_i$ but the index variables j, k, h, and let V_{r_i} be the set of all variables in r_i. Obviously, $V_{\hat{r}_i} \subseteq V_{r_i}$. We set $\phi(V_{\hat{r}_i}) = \hat{\phi}(V_{\hat{r}_i})$. Let us now consider the nodes of DT corresponding to the supplementary counting predicate (if present) and to the counting facts of P_ν, $0 \leq \nu \leq p-1$. We denote the solving assignments for these nodes by spcnt.ϕ and cnt.ϕ_ν, $0 \leq \nu \leq p-1$ respectively. The following three cases are possible:

Case 1: M_Q is not reduced. Hence, the body of $\hat{r}_i$ contains the predicate spcnt.$R^S.r_i(j, k, h, z)$. Since all variables in $V_{r_i} - V_{\hat{r}_i}$ appear in the supplementary counting rule, we can set $\phi(V_{r_i} - V_{\hat{r}_i}) = \text{spcnt}.\phi(V_{r_i} - V_{\hat{r}_i})$. So, we have defined an assignment for all variables in r_i. We have that r_i differs from $\hat{r}_i$ only because the datum predicates $Q_1, \ldots, Q_r$ are replaced with the supplementary counting predicate and because the c-predicates in r' (both in the head and in the body) have three additional arguments (i.e., the index arguments). Then, after replacing every variable y with $\phi(y)$, the c-predicates in the body of r_i and the datum predicates $W_1, \ldots, W_q$ are equal to the corresponding children of $R(\boldsymbol{b})$, and the head c-predicate is equal to $R(\boldsymbol{b})$. Furthermore, since the supplementary variables in the rule $\hat{r}_i$ and in the supplementary counting rule are the same by construction, it is easy to see that, for

each variable x in the supplementary counting predicate of $\hat{r}_i$, $\hat{\phi}(x) = \text{spcnt}.\phi(x)$. Hence, also the datum predicates $Q_1, \ldots, Q_r$ in the body of r_i are equal to the corresponding child nodes of $R(\boldsymbol{b})$ after variable replacements. It thus follows that $R(\boldsymbol{b})$ is solved in r_i and therefore, it is inferred from $\text{LP} \cup D$.

Case 2: M_Q is reduced and either there are unreduced variables in the rule r_i or r_i contains more than one c-predicate in its body (i.e., $p > 1$). We need the following claim.

Claim 4.4. *For every* ν, $1 \leq \nu \leq p-1$, $\text{cnt}.\phi_0 = \text{cnt}.\phi_\nu$ *and if the supplementary counting predicate exists in the body of* $\hat{r}_i$, $\text{cnt}.\phi_0 = \text{spcnt}.\phi$.

Proof. By definition of reduced binding graph, we have that

(a) R^S is strongly reduced, and

(b) no database predicate in r_i is solved by B_S, where B_S is the set of all variables appearing in the head of r_i.

Therefore, by Lemma 4.2, there is exactly one counting fact with indices $\hat{j}, \hat{k}, \hat{h}$ and predicate symbol $\text{cnt}.R^S$, say $\text{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{c})$. Consider now the nodes in DT corresponding to the supplementary counting predicate (if present) and to the counting predicates of $P_0, \ldots, P_{p-1}$. Since the counting rules and the supplementary counting rule defining these predicates have the same body by construction, the children of each of the above nodes are a counting fact with indices $\hat{j}, \hat{k}, \hat{h}$ and with predicate symbol $\text{cnt}.R^S$, i.e., $\text{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{c})$ and r nodes corresponding to the solved datum predicates $Q_1, \ldots, Q_r$ $(r \geq 0)$. Since no database predicate in r_i is solved by B_S, if $Q_1, \ldots, Q_r$ are present in r_i, then they are comparison predicates. It is then easy to see that all of the above nodes have the same children and, therefore, the statement of the claim holds. □

We continue the proof of the lemma by setting $\phi(V_{r_i} - V_{\hat{r}_i}) = \text{cnt}.\phi_0(V_{r_i} - V_{\hat{r}_i})$. We now show that ϕ is a solving assignment for $R(\boldsymbol{b})$ in r_i. Obviously, after replacing every variable y with $\phi(y)$, the c-predicate P_0 and the datum predicates $W_1, \ldots, W_q$ are equal to the corresponding children of $R(\boldsymbol{b})$. Let us now consider all other predicates in r_i:

(a): the head predicate $R(\boldsymbol{t})$ or a datum predicate Q_t, $1 \leq t \leq r$, in the body of r_i. If all variables in Q_t and in the S-arguments of $R(\boldsymbol{t})$ are in $V_{r_i} - V_{\hat{r}_i}$, then it is immediate to see that Q_t equals the corresponding child of $R(\boldsymbol{b})$, and $R(\boldsymbol{t})$ equals $R(\boldsymbol{b})$ after replacing every variable y in r_i with $\phi(y)$. Suppose now that some variable of Q_t or of the S-arguments of $R(\boldsymbol{t})$, say x, appears in $V_{\hat{r}_i}$. Then x is an unreduced variable and appears as an argument of the supplementary counting predicate of $\hat{r}_i$. We have that $\hat{\phi}(x) = \text{spcnt}.\phi(x)$ by the definition of a derivation tree and $\text{spcnt}.\phi(x) = \text{cnt}.\phi_0(x)$ by Claim 4.4. Hence, since $\hat{\phi}(x) = \phi(x)$ by construction, $\phi(x) = \text{cnt}.\phi_\nu(x)$ and again Q_t equals the corresponding child of $R(\boldsymbol{b})$, and $R(\boldsymbol{t})$ equals $R(\boldsymbol{b})$ after replacing every variable y in r_i with $\phi(y)$.

(b): a c-predicate P_ν, $0 \leq \nu \leq p-1$. Consider any variable x in the T_ν-arguments of P_ν. If x is in $V_{r_i} - V_{\hat{r}_i}$, then $\phi(x) = \text{cnt}.\phi_0(x)$. But either $\nu = 0$ or $\text{cnt}.\phi_0(x) = \text{cnt}.\phi_\nu(x)$ by Claim 4.4, so $\phi(x) = \text{cnt}.\phi_\nu(x)$. On the other hand, if x is in $V_{\hat{r}_i}$, then

x is an unreduced variable and appears as an argument of the supplementary counting predicate of $\hat{r}_i$. We have that $\phi(x)=\text{spcnt}.\phi(x)$ by the definition of a derivation tree and $\text{spcnt}.\phi(x)=\text{cnt}.\phi_0(x)=\text{cnt}.\phi_{\nu}(x)$ by Claim 4.4. Since $\hat{\phi}(x)=\text{cnt}.\phi_0(x)$ by construction, $\phi(x)=\text{cnt}.\phi_0(x)$, and then P_{ν} equals the corresponding child of $R(\boldsymbol{b})$ after replacing every variable y in r_i with $\phi(y)$.

In sum, ϕ is a solving assignment for $R(\boldsymbol{b})$ in r_i and, therefore, $R(\boldsymbol{b})$ is inferred from $\text{LP}\cup D$.

Case 3: M_Q is reduced and there are no unreduced variables in r_i and r_i contains exactly one c-predicate in its body. Hence, every variable x that appears in $Q_1,\ldots,Q_r$, in the S-arguments of $R(\boldsymbol{t})$, or in the T_0-arguments of P_0 is in $V_{r_i}-V_{\hat{r}_i}$; so $\phi(x)=\text{cnt}.\phi_0(x)$. It follows that ϕ is a solving assignment for $R(\boldsymbol{b})$ in r_i and, therefore, $R(\boldsymbol{b})$ is inferred from $\text{LP}\cup D$. This concludes the proof. □

Lemma 4.5. *Let $Q=\langle G, \text{LP}, D\rangle$ be a query that has the binding-passing property and whose binding graph is M_Q. Let $\bar{Q}=\langle G, \overline{\text{LP}}, D\rangle$ be the modified query produced by the Generalized Counting Method. Let $\text{cnt}.R^S(\hat{j},\hat{k},\hat{h},\boldsymbol{a})$ be a counting fact that is inferred from $\overline{\text{LP}}\cup D$ and let $R(\boldsymbol{b})$ be a c-fact that is inferred from $\text{LP}\cup D$ such that $\boldsymbol{b}^S=\boldsymbol{a}$. Then the c-fact $R^S(\hat{j},\hat{k},\hat{h},\boldsymbol{c})$ is inferred from $\overline{\text{LP}}\cup D$, where $\boldsymbol{b}=\boldsymbol{c}$ if M_Q is not reduced, or $\boldsymbol{b}^{S^-}=\boldsymbol{c}$ if M_Q is reduced.*

Proof. We carry out the proof by induction on the number of c-facts in the derivation tree DT of $R(\boldsymbol{b})$.

Basis of the induction: There is only one c-fact in DT. Then this c-fact is the root $R(\boldsymbol{b})$ that is labelled by an exit rule in LP, say the following rule r_i

$$R(\boldsymbol{u}) \text{ :- } W_1,\ldots,W_q.$$

The corresponding modified exit rule in $\overline{\text{LP}}$ is the following rule $\hat{r}_i$:

$$R^S(j,k,h,\boldsymbol{u}) \text{ :- } \text{cnt}.R^S(j,k,h,\boldsymbol{x}), W_1,\ldots,W_q.$$

We construct a derivation tree $\overline{\text{DT}}$ for $R^S(\hat{j},\hat{k},\hat{h},\boldsymbol{c})$ as follows. $R^S(\hat{j},\hat{k},\hat{h},\boldsymbol{c})$ is the root and is labelled by $\hat{r}_i$. The facts corresponding to $W_1,\ldots,W_q$ in DT are added to $\overline{\text{DT}}$ as children of the root. Moreover, the non-leaf child $\text{cnt}.R^S(\hat{j},\hat{k},\hat{h},\boldsymbol{a})$ is added. Since this fact is inferred from $\overline{\text{LP}}\cup D$ by hypothesis and the leaf facts are obviously in D, we only have to show that there is a solving assignment for $R^S(\hat{j},\hat{k},\hat{h},\boldsymbol{c})$ in $\hat{r}_i$. But r_i and $\hat{r}_i$ have the same variables (besides indices); so the solving assignment for $R(\boldsymbol{b})$ in r_i is also a solving assignment for $R^S(\hat{j},\hat{k},\hat{h},\boldsymbol{c})$ in $\hat{r}_i$.

Inductive step: the lemma holds whenever DT has less than s c-facts, where $s>1$ (inductive hypothesis). We prove that it also holds when DT has s c-facts. Say that the root of DT is labelled by r_i where r_i is the following recursive rule:

$$R(\boldsymbol{t}) \text{ :- } Q_1,\ldots,Q_r, P_1,\ldots,P_{p-1}, W_1,\ldots,W_q.$$

The corresponding modified recursive rule in $\overline{\text{LP}}$ is the following rule $\hat{r}_i$:

$$R^S(j,k,h,\boldsymbol{u}) \text{ :- } \text{spcnt}.R^S.r_i(j,k,h,\boldsymbol{z}), \hat{P}_0,\ldots,\hat{P}_{p-1}, W_1,\ldots,W_q.$$

The supplementary counting predicate (if present) is defined by the following supplementary counting rule:

$$\text{spcnt}.r_i.R^S(j, k, h, z) \ \text{:-}\ \text{cnt}.R^S(j, k, h, \boldsymbol{x}), Q_1, \ldots, Q_r.$$

The counting set associated to every $\hat{P}_\nu$, $0 \leq \nu \leq p-1$, is defined by the following recursive counting rule:

$$\text{cnt}.P_\nu^{T_\nu}(j+1, m \times k + i, p \times h + \nu, \boldsymbol{y}) \ \text{:-}\ \text{cnt}.R^S(j, k, h, \boldsymbol{x}), Q_1, \ldots, Q_r.$$

Hence, it is easy to see that, using the solving assignment ϕ for $R(\boldsymbol{b})$ and the hypothesis that $\text{cnt}.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{a})$ is inferred from $\text{LP} \cup D$, it is possible to infer the facts

$$\text{spcnt}.r_i.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{d}) \quad \text{and} \quad \text{cnt}.P_\nu^{T_\nu}(\hat{j}+1, m \times \hat{k} + i, p \times \hat{h} + \nu, \boldsymbol{e})$$

such that, for each P_ν, $0 \leq \nu \leq p-1$, $\boldsymbol{f}^{T_\nu} = \boldsymbol{e}$, where $\boldsymbol{f}$ are the arguments of the node in DT corresponding to P_ν. Hence, by inductive hypothesis, the c-facts $P_\nu^{T_\nu}(\hat{j}+1, m \times \hat{k} + i, p \times \hat{h} + \nu, \boldsymbol{g})$, where $\boldsymbol{f}^{T_\nu} = \boldsymbol{g}$, are inferred from $\overline{\text{LP}} \cup D$. We can now construct a derivation tree $\overline{\text{DT}}$ for $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{c})$ using the supplementary fact $\text{spcnt}.r_i.R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{d})$, the c-facts $P_\nu^{T_\nu}(\hat{j}+1, m \times \hat{k} + i, p \times \hat{h} + \nu, \boldsymbol{g})$, and the nodes of DT corresponding to $W_1, \ldots, W_q$. We have that every variable in $\hat{r}_i$ (except indices) is also in r_i. Moreover, all children of $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{c})$ were generated using the solving assignment ϕ for $R(\boldsymbol{b})$. So ϕ is also a solving assignment for $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{c})$. Finally, all children of this root are either in D or are inferred from $\overline{\text{LP}} \cup D$. It follows that $\overline{\text{DT}}$ is a derivation tree, and then $R^S(\hat{j}, \hat{k}, \hat{h}, \boldsymbol{c})$ is inferred from $\overline{\text{LP}} \cup D$. □

Theorem 4.6. *Let $Q = \langle G, \text{LP}, D \rangle$ be a query that has the binding-passing property and let $\bar{Q} = \langle G, \overline{\text{LP}}, D \rangle$ be the modified query produced by the Generalized Counting Method. Then Q and $\bar{Q}$ are equivalent.*

Proof. Let M_Q be the binding graph of Q and let $\boldsymbol{a}$ be the bound arguments in G, denoted by S. We first show that any answer of $\bar{Q}$, say $\hat{G}(\boldsymbol{b})$, is also an answer of Q. Considering the goal rule in $\overline{\text{LP}}$, also $G^S(0, 0, 0, \boldsymbol{c})$ is inferred from $\overline{\text{LP}} \cup D$, where $\boldsymbol{b} = \boldsymbol{c}$ if M_Q is not reduced, or $\boldsymbol{b}^S = \boldsymbol{a}$ and $\boldsymbol{b}^{S^-} = \boldsymbol{c}$ if M_Q is reduced. By Lemma 4.3, $G(\boldsymbol{b})$ is inferred from $\text{LP} \cup D$, i.e., it is an answer of Q.

Let us now prove that any answer $\hat{G}(\boldsymbol{b})$ of Q is also an answer of $\bar{Q}$. Obviously, $\text{cnt}.G^S(0, 0, 0, \boldsymbol{a})$ is inferred from $\overline{\text{LP}} \cup D$. Hence, by Lemma 4.5, $G^S(0, 0, 0, \boldsymbol{c})$ is inferred from $\overline{\text{LP}} \cup D$, where $\boldsymbol{b} = \boldsymbol{c}$ if M_Q is not reduced, or $\boldsymbol{b}^S = \boldsymbol{a}$ and $\boldsymbol{b}^{S^-} = \boldsymbol{c}$ if M_Q is reduced. Considering the goal rule, also $G(\boldsymbol{b})$ is inferred from $\overline{\text{LP}} \cup D$, i.e., it is an answer of $\bar{Q}$. □

The next result shows that counting and modified rules can be generated efficiently.

Proposition 4.7. *Let $Q = \langle G, \text{LP}, D \rangle$ 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 $\bar{Q} = \langle G, \overline{\text{LP}}, D \rangle$ in time linear in the size of* LP *and* G.

Proof. Clearly, once the binding graph has been constructed and it has been checked whether or not it is reduced, every rule in $\overline{\text{LP}}$ can be generated in time linear in the size of some rule in LP or of G. Moreover, the same rule in LP can be used at most $O(2^k)$ times, where k is the maximum predicate arity in LP, to generate rules in $\overline{\text{LP}}$. Hence, $\bar{Q}$ can be constructed in time $O(2^k \times s + g)$ time, where s and g are the size of LP and G respectively. If k is bound, then the generalized counting method works in linear time in s and g. □

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 [7] or of a particular class of queries [13]. 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 computations 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 [5, 6, 19]). 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 [11]). However, the Generalized Counting Method also allows us to use different implementations of joins (including, e.g., a sort-merge join) 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, Fig. 7(a)).

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 Figs. 7(a) and 7(b)).

Shared solved predicates

Counting rules and supplementary counting rules might share the same solved predicates. For instance, in Fig. 7, 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. Then, the counting and special counting predicates can simply be derived from the allcnt predicate 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 that is required is that these predicates can be solved independently of the recursive strong component under consideration. For instance, the technique presented in [29] can be used to deal effectively with *nonrecursive rules*, possibly containing function symbols. Said technique provides 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 strong component as our c-predicates) can be used as solved datum predicates. This is tantamount to determining whether the corresponding goal in the rule can be solved for the given set of bindings. To this end, we can again apply the Generalized Counting Method. Take, for instance, a query G: $\mathrm{MG}(L_1, L_2, X)$, defined against a logic program consisting of the rules of Figs. 2 and 3 combined. Then, in order to solve this query, we shall 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 Fig. 9 (since we only have one recursive rule we only use one index).

Finally, we need to link the rules of Fig. 9 with the last counting rule of Fig. 7(b). This can, for instance, be accomplished by redefining the goal $x < x_1$ of Fig. 7(b) as follows:

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

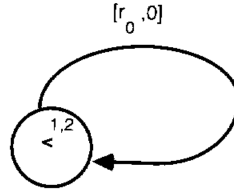
(This is a rather coarse solution, presented here only as a quick illustration on how things could function; a more refined solution is given in [8].)

Trivial modified rules

It is easy to see that the only function of the modified recursive rule in Fig. 9, is to decrement the index to zero one step at a time. We can thus dispense with this rule and write a new modified goal:

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

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.$<^{1,2}(j, x, s(x))$.
$<^{1,2}(j-1)$:- $<^{1,2}(j)$.
$\bar{G}$: $<^{1,2}(0)$?

Fig. 9. Implementation of the "less-than" rules of Fig. 3.

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

Symmetrically, it is 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 [1].

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 our methods terminate. Note that the Generalized Counting Method recasts the original query Q into two fixpoint computations: the *counting set computation* and the *modified c-predicate computation*. Whenever both these computations terminate in a finite number of steps, we shall say that the *Generalized Counting Method is safe w.r.t. to the query Q*.

The first condition for a query to be safe is that every step of the fixpoint computation can be carried out in a finite amount of time, thus the relational algebra expression associated to a single rule is effectively computed. In this case, we say that a query is *single-step safe*. In dealing with the problem of single-step safety we

see that we have to handle two possible sources of unsafe behaviour. One is *dangling variables*, i.e., variables appearing in the head and not in the body. The second is datum predicates, such as comparison predicates, that are safely solved only when certain arguments are bound (obviously, database predicates corresponding to finite database relations are always safely solved). Sufficient conditions for the solvability of rules containing comparison predicates were given in Section 3.1 in the course of the binding propagation analysis. Extensions of those conditions to other kinds of datum predicates, such as derived predicates, is possible but outside the scope of this paper. Here, instead, we give the conditions that allow us to infer the safety of the computation prescribed by the counting method from the fact that the only datum predicates are (finite) database predicates and comparison predicates.

Let r_i be a rule in LP and let I be an index set denoting arguments in the head predicate of r_i. We say that r_i is *solved by* I if all variables of r_i are bound by $B^I \cup B_c \cup B_d$, where B^I are all variables appearing in the I-arguments of the head predicate and B_c (respectively B_d) are the variables of all c-predicates (respectively, database predicates) in the body of r_i. For instance, given $I = \{1\}$, all the rules of Figs. 1 and 6 as well as the recursive rules of Fig. 2 and the exit rule of Fig. 4 are solved by I whereas the exit rules of Fig. 2 and the recursive rule of Fig. 4 are not.

The following result directly derives from the definition of variables made bound by a set of variables (see Section 3.1).

Fact 5.1. *Given a query* $Q = \langle G, \text{LP}, D\rangle$, *if every rule in* LP *is solved by* $S = \emptyset$, *then* Q *is single-step safe.*

We next show that the conditions for safety for the query generated by the Generalized Counting Method are much less restrictive, and that this is single-step safe in many cases when the original query is not. A query Q is said to be *solved* if it has the binding-passing property and, for each node R^S of its binding graph, each (recursive or exit) rule r_i such that the predicate symbol of its head is R is *solved* by S. Note that a solved query is not necessarily single-step safe. For instance, the query $\text{MG}^{1,2}$ on the logic program of Fig. 2 is solved but not single-step safe. On the other hand, the corresponding query generated by the Generalized Counting Method is single-step safe. An example of a nonsolved query is given by the query R^1 on the logic program of Fig. 4.

Proposition 5.2. *Let* $Q = \langle G, \text{LP}, Q\rangle$ *be a query with the binding-passing property and* $\bar{Q}$ *the modified query constructed by the Generalized Counting Method. Then,*

(a) *deciding whether or not* Q *is solved can be done in time linear in the size of* LP *and* G;

(b) *if* Q *is solved, then* $\bar{Q}$ *is single-step safe.*

Proof. (a): We observe that the algorithm, presented in the proof of Proposition 3.1 for finding solved datum predicates, can be easily extended to perform the test.

(b): We consider the two sets of rules generated by the Generalized Counting Method as two separated logic programs $\overline{\mathrm{LP}}_1$ and $\overline{\mathrm{LP}}_2$, corresponding to the counting set computation and the modified c-predicate computation respectively. We observe that counting predicates and supplementary counting predicates are c-predicates in $\overline{\mathrm{LP}}_1$; moreover, they are datum predicates in $\overline{\mathrm{LP}}_2$. Let us first of all consider any (counting or supplementary counting) rule $\hat{r}_i$ in $\overline{\mathrm{LP}}_1$. Let $\hat{r}_i$ be associated to an arc leaving a node R^S, labelled by a rule r_i in LP. Since r_i is solved by S by hypothesis and the S-arguments appear in the counting predicate in the body of $\hat{r}_i$, it is easy to see that the rule $\hat{r}_i$ is solved by $B^S \cup B_{\mathrm{d}}$. But B^S are all variables appearing in the c-predicate in the body of $\hat{r}_i$, i.e., the counting predicate. Hence, $\hat{r}_i$ is solved by $S = \emptyset$. It follows that $\overline{\mathrm{LP}}_1$ is single-step safe by Fact 5.1. As a consequence, at every (finite) step of the overall fixpoint computation of $\overline{\mathrm{LP}}$, the derived relations corresponding to the counting and supplementary counting predicates are finite. Therefore, such predicates can be thought of as finite database predicates in $\overline{\mathrm{LP}}_2$. Consider now a modified exit rule $\hat{r}_i$ in $\overline{\mathrm{LP}}_2$ that is associated to an arc leaving a node R^S and to an exit rule r_i in LP. By hypothesis, r_i is solved by S. Since all variables in the S-arguments of the head predicate of r_i appear in the counting predicate in the body of $\hat{r}_i$, it is easy to see that $\hat{r}_i$ is solved by $I = \emptyset$. Let $\hat{r}_i$ be now a modified recursive rule in $\overline{\mathrm{LP}}_2$ that is associated to an arc leaving the node R^S, whose label is a recursive rule r_i in LP. Again, by hypothesis, r_i is solved by S, i.e., all variables of r_i are bound by $B^S \cup B_{\mathrm{c}} \cup B_{\mathrm{d}}$. Let $B_{\hat{\mathrm{c}}}$ and $B_{\hat{\mathrm{d}}}$ be the set of all variables appearing in the c-predicates and in the database predicates of $\hat{r}_i$ respectively. By construction, all variables of r_i as well as those of $\hat{r}_i$ are bound by $B^S \cup B_{\hat{\mathrm{c}}} \cup B_{\hat{\mathrm{d}}}$. On the other hand, we observe that if a variable x in B^S is necessary to bind some variable in $\hat{r}_i$, then x is in the supplementary counting predicate of $\hat{r}_i$, i.e., in $B_{\hat{\mathrm{d}}}$. Therefore, $\hat{r}_i$ solved by $I = \emptyset$. By Fact 5.1, $\bar{Q}$ is a single-step safe. □

From now on, we only consider solved queries as input to the Generalized Counting Method. The following property follows immediately from the definitions.

Proposition 5.3. *The Generalized Counting Method is safe w.r.t. a solved query if and only if the counting set fixpoint computation converges in a finite number of steps.*

Proof. At every step of the modified c-predicate fixpoint computation, the three indices are decreased. Moreover, since the query is solved, by Proposition 5.2, every step is executed in a finite amount if time. Hence, this computation terminates after a finite number of steps. □

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 $t=f(t_1,\ldots,t_k)$, 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, $|x\cdot x|=|x|+|x|+1=2|x|+1$. In general, there is no information on the actual length of x, except that $|x|\geqslant 1$. Thus $|x\cdot x|\geqslant 3$.

The length of a set of terms S is the sum of the lengths of all terms in S. For instance, the length of the bound arguments (i.e., $x\cdot y$, $x_1\cdot y_1$) in rule r_0 of the MG example in Fig. 7 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 $[r_i, \nu]$. The length balance associated with this arc is defined as the 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 νth c-predicate in the body (i.e., the arguments denoted by T). For instance, the length balance for the arc labelled $[r_0, 0]$ in the binding graph of Fig. 5(b) 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 1 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 5.4. *Let Q be a solved query. If the length balance associated with every cycle in the binding graph of Q is positive, then the Generalized Counting Method is safe w.r.t. Q.*

Proof. Let s be the length of the arguments in the predicate of the counting exit rule. The first rule fired by the counting set fixpoint computation is the counting exit rule. Since this rule is actually a fact, the argument length of the counting predicate is a constant, say s. Every step of the fixpoint computation is carried out in a finite amount of time because of Proposition 5.2. Furthermore, at every cycle of the recursion, the length s is decreased because of the condition of the theorem. Hence, the recursion of the counting set fixpoint computation is "well-founded" in

the sense that some argument length will eventually become 0 and no more recursion cycles will be possible. It follows that the counting set fixpoint computation terminates. By Proposition 5.3, the Generalized Counting Method is safe w.r.t. Q. □

Thus, the Generalized Counting Method is safe w.r.t. the query $MG^{1,2}$ on the logic program of Fig. 2 and w.r.t. the query $<^{1,2}$ on the logic program of Fig. 3.

We note that testing the condition of Theorem 5.4 may require exponential time in the size of the binding graph. However, this is not a big problem since binding graphs have a very small number of cycles in many practical cases. The real limitation is the actual applicability of the theorem since there are many cases where more elaborated or completely different techniques for testing safety must be used. Neverthelesss, our belief is that Theorem 5.4 can be very useful in many typical situations of recursive rules with function symbols, such as appending two lists and searching and manipulating trees and lists.

A simple extension of Theorem 5.4 is the following. 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. It is easy to see that this is also a sufficient condition for safety. More complex situations arise when 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 these cases is given in [23, 25]. For variables that belong to some database predicate, it is often reasonable to assume that their length is 1. 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 solumn:

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

$$P(b \cdot b \cdot x) \text{ :- } Q(x, y), P(x \cdot y).$$

$$P(b).$$

Finally, there are situations such as those of examples of Figs. 1 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 [17]. 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 [5] and minimagic method [18] 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 duplications. If duplicates show up, then one will fall back on the magic set method. This hybrid approach, known as *magic counting* is described in [17, 19].

6. Conclusion

We have presented a new method, named generalized counting, that is very efficient (see the evaluation in [7, 13]) 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 nonlinear recursion.

The paper also discussed the application of the method to solve nested recursive predicates (a further extension of the Generalized Counting Method for handling such kind of queries is given in [8]). 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.

Acknowledgment

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

References

[1] A.V. Aho and J. Ullman, Universality of data retrieval languages, in: *Proc. POPL Conference*, San Antonio TX (1979) 110–120.
[2] L. Aiello and Cecchi, Adding a closure operator to the extended relational algebra . . . , Tech. Rep., University of Rome, 1985.
[3] F. Bancilhon, Naive evaluation of recursively defined relations, in: *On Knowledge Base Managemant System—Integrating Database and AI Systems* (Springer, Berlin, 1985).
[4] R. Bayer, U. Guntzer and W. Kiessling, On the evaluation of recursion in deductive DB systems by efficient differential fixpoint iteration, Tech. Rep., Technische Univ. Munich, 1985.
[5] F. Bancilhon, D. Maier , Y. Sagiv and J. Ullman, Magic sets and other strange ways to implement logic programs, in: *Proc 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems* (1986) 1-15.
[6] F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, Magic sets: algorithms and examples, Unpublished manuscript, 1985.
[7] F. Bancilhon and R. Ramakrishan, An amateur's introduction to recursive query processing strategies, in: *Proc. ACM SIGMOD Internat. Conf. on Management of Data*, Washington, DC (1986) 16–52.
[8] C. Beeri and R. Ramakrishnan, On the power of magic, in: *Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems* (1987) 269–283.
[9] A.K. Chandra and D. Harel, Horn clauses and the fixpoint hierarchy, in: *Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems* (1982) 158–163.

[10] G. Gardarin and DeMaindreville, Evaluation of database recursive logic programs as recursive function series, in: *Proc. ACM SIGMOD Internat. Conf. on Management of Data*, Washington, DC (1986) 177-186.
[11] L.J. Henschen and S.A. Naqvi, On compiling queries in recursive first-order databases, *J. ACM* **31**(1) (1984) 47-85.
[12] E.L. Lozinskii, A problem-oriented inferential database system, *ACM TODS* **11**(3) (1986) 323-356.
[13] A. Marchetti-Spaccamela, A. Pelaggi and D. Saccà, Worst-case complexity analysis of methods for logic query implementation, in: *Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems* (1987) 294-301.
[14] D. McKay and S. Shapiro, Using active connection graphs for reasoning with recursive rules, in: *Proc. 7th IJCAI* (1981) 368-374.
[15] S. Parker et al., Logic programming and databases, in: L. Kerschberg, ed., *Expert Database Systems* (Benjamin/Cummings, Menlo Park, CA, 1986).
[16] R. Reiter, On closed world databases, in: H. Gallaire and J. Minker, eds., *Logic and Databases* (Plenum, New York, 1978) 55-76.
[17] D. Saccà and C. Zaniolo, On the implementation of a simple class of logic queries for databases, in: *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems* (1986) 16-23.
[18] D. Saccà and C. Zaniolo, Implementation of recursive queries for a data language based on pure Horn logic, MCC Tech. Rep. 092-86, 1986.
[19] D. Saccà and C. Zaniolo, Magic counting methods, in: *Proc. ACM SIGMOD Conf.* (1987).
[20] A. Tarski, A lattice theoretical fixpoint theorem and its application, *Pacific J. Math.* **5** (1955) 285-309.
[21] J.D. Ullman, *Principles of Database Systems* (Computer Science Press, Rockville, MD, 1982).
[22] J.D. Ullman, Implementation of logical query languages for databases, *TODS* **10**(3) (1985) 289-321.
[23] J.D. Ullman and A. Van Gelder, Testing applicability of top-down capture rules, Rep. STAN-CS-85-1046, Stanford University, 1985.
[24] M.H. Van Emden and R. Kowalski, The semantics of predicate logic as a programming language, *J. ACM* **23**(4) (1976) 733-742.
[25] A. Van Gelder, A message passing framework for logical query evaluation, in: *Proc. ACM SIGMOD Internat. Conf. on Management of Data*, Washington, DC (1986) 155-165.
[26] L. Vieille, Recursive axioms in deductive databases: the query-subquery approach, in: *Proc. First Internat. Conf. on Expert Database Systems*, Charleston, SC (1986).
[27] C. Zaniolo, Prolog: a database query language for all seasons, in : L. Kerschberg, ed., *Expert Database Systems* (Benjamin/Cummings, Menlo Park, CA, 1986).
[28] C. Zaniolo, The representation and deductive rretrieval of complex objects, in: *Proc. 11th VLDB* (1985) 459-469.
[29] C. Zaniolo, Safety and compilation of non-recursive Horn clauses, in: *Proc. First Internat. Conf. on Expert Database Systems*, Charleston, SC (1986).