A Framework for Testing Safety and Effective Computability*

RAVI KRISHNAMURTHY†
Hewlett-Packard Laboratories, Palo Alto, California

RAGHU RAMAKRISHNAN†
University of Wisconsin-Madison, Madison, Wisconsin 53706

AND

ODED SHMUELÍ
Technion, Israel

Received May 29, 1991; revised May 9, 1995

This paper presents a methodology for testing a logic program containing function symbols and built-in predicates for safety and effective computability. Safety is the property that the set of answers for a given query is finite. A related issue is whether the evaluation strategy can effectively compute all answers and terminate. We consider these problems under the assumption that queries are evaluated using a fair bottom-up fixpoint computation. We also model the use of function symbols, to construct complex terms such as lists, and arithmetic operators, by considering Datalog programs with infinite base relations over which finiteness constraints and monotonicity constraints are imposed. One of the main results of this paper is a recursive algorithm, check_clique, to test the safety and effective computability of predicates in arbitrarily complex cliques in the predicate connection graph. This algorithm takes certain procedures as parameters, and its applicability can be strengthened by making these procedures more sophisticated. We specify the properties required of these procedures precisely, and present a formal proof of correctness for the algorithm check_clique. This work can be seen as providing a framework for testing safety and effective computability of recursive programs, in some ways analogous to the capture rules framework of Ullman. A second important contribution is a framework for analyzing programs that are produced by the Magic Sets transformation utilizing check_clique to analyze recursive programs. The transformed program unfortunately often has a clique structure that combines several cliques of the original program. Given the complexity of algorithm check_clique, it is important to keep cliques as small as possible. We deal with this problem by considering cliques in an intermediate program, called the adorned program, produced by the Magic Sets transformation. The clique structure of the adorned program is similar to that of the original program, and by showing how to analyze the transformed program in terms of the cliques in the adorned program, we avoid the potentially expensive analysis of the cliques in the transformed program. © 1996 Academic Press, Inc.

1. INTRODUCTION

The evaluation of recursive queries expressed as sets of Horn clauses over a database has recently received much attention. Consider the following program:

\[
\text{anc}(X, Y) :- \text{par}(X, Y).
\]

\[
\text{anc}(X, Y) :- \text{par}(X, Z), \text{anc}(Z, Y).
\]

and let the query be

\[ \text{Query: } \text{anc}(\text{john}, Y) \land \]?

Assume that a database contains a parenthood relation \( \text{par} \). Then the program defines a derived relation describing ancestors, and the query asks for the ancestors of \( \text{john} \). This can be evaluated as a Prolog program, but there are two drawbacks to doing so. First, Prolog uses a top-down backtracking strategy that is incomplete. That is, it may not produce all answers implied by reading the rules as statements in standard logic. For example, Prolog will loop forever without producing any answers if the \( \text{anc} \) relation is expressed equivalently as

\[
\text{anc}(X, Y) :- \text{par}(X, Y).
\]

\[
\text{anc}(X, Y) :- \text{anc}(X, Z), \text{par}(Z, Y).
\]

The second problem with Prolog is that its evaluation strategy always computes joins by a nested loop join technique that is inefficient in the presence of a large number of facts [4].

* A preliminary version of the results in Sections 1–6 was presented, without proofs, at “ACM PODS 87.”
† Most of this work was done while at MCC, Austin.
‡ University of Wisconsin-Madison. Work partly done while visiting MCC, and partially supported by an IBM Faculty Development Award and NSF grant IRI-8804319.
§ Technion, Israel. Work done while at MCC and partially supported by the Fund for Promotion of Research at the Technion.


All rights of reproduction in any form reserved.

100
This has motivated the development of alternative evaluation methods based on bottom-up evaluation, which is a well-known strategy for evaluating logic programs. It serves to define the least fixed point semantics, and is known to be complete [8]. The major problem with this approach is that it does not restrict the computation by utilizing constants in the query. For example, in the previous query, only the ancestors of john are needed, but simple bottom-up evaluation would compute the entire ancestor relation and then select john’s ancestors. Several refinements of bottom-up evaluation and other strategies have been proposed to deal with this problem [3]. The main thrust of these strategies is to improve efficiency by restricting the computation to tuples that are related to the query.

The Magic Sets strategy [2] restricts computation by rewriting the program so that it only computes facts relevant to the query. The original version achieved this restriction only for certain kinds of rules (essentially, linear recursive rules). An extended version, Generalized Magic Sets, which is based on the notion of sideways information passing graphs, achieves this restriction for all programs in which variables that appear in the head of a rule also appear in the body [5].

Since bottom-up strategies promise to be an efficient approach for evaluating recursive queries, it is important to investigate their properties. In the context of databases, the set of answers is always finite in the absence of recursion and negation. It is widely expected that while the addition of recursion to the query language will make it more expressive, the typical query will yield a finite set of answers. Thus, an important question is whether the set of answers for a given query is indeed finite. If not, the program is probably incorrect. This is the safety problem and has been shown to be undecidable for Horn clause programs with function symbols [11, 7]. A related issue is whether the evaluation strategy (in particular, bottom-up evaluation, possibly after rewriting the program) computes all answers and terminates.

While the question of safety and termination are important questions for general logic programs, they are particularly of interest for program expressing database queries. We address these questions in this paper. We consider programs that do not contain terms constructed using function symbols (i.e., all arguments are constants or variables), also known as Datalog. However, we allow infinite base relations, and these can be used to model terms constructed with function symbols and evaluate functions such as arithmetic operations. We use the name Extended Datalog to denote Datalog with this extension.

Some results relating to these problems have been presented in the context of testing the applicability of top-down capture rules [10, 14, 1]. These problems were also addressed in [9]. In this paper, we make use of, and extend, this earlier work.

One of the main results of this paper is a recursive algorithm, check clique, for testing the safety and effective computability of predicates in arbitrarily complex cliques. This algorithm takes certain procedures as parameters, and its applicability can be strengthened by making these procedures more sophisticated. We specify the properties required of these procedures precisely, and present a formal proof of correctness for the algorithm check clique. While the complexity of this algorithm is exponential in the size of the clique, we believe that most cliques in practice are likely to be sufficiently small that this is acceptable, even if the program is large.

This work can be seen as providing a framework for testing safety and effective computability of recursive programs, in some ways analogous to the capture rules framework of Ullman [12]. The capture rules framework considers the program represented as a rule/goal graph and allows the design of capture rules that describe how nodes in the graph can be “captured.” Intuitively, a node can be captured if an effective procedure can be constructed to compute the predicate (or rule) denoted by that node. A capture rule states that a given node may be captured if certain other nodes in the graph have already been captured and is substantiated by a procedure that computes the node given procedures for capturing these other nodes. Thus, if the query node in the rule/goal graph can be captured, we know that it is safe and effectively computable.

We proceed by assuming a bottom-up model of execution and present algorithms for testing safety and effective computability of programs in this model. Thus, if a query is declared to be safe and effectively computable, bottom-up execution according to our model is an effective procedure for computing it. An important aspect of our approach to testing is that it is based on a clique-by-clique analysis. We develop algorithms for testing the safety of predicates in a clique. We test a given program by topologically sorting its cliques and then testing them in topological order starting from the leaf cliques. This makes the analysis of large programs tractable since the complexity of our algorithms is dominated by the size of recursive cliques, and we expect these to be small, independent of the size of the program.

Thus, our framework is similar to the capture rules framework in the following sense: If a query is declared to be safe and effectively computable (analogous to “captured”), there is a known way to effectively compute it. The analogy goes further. Both frameworks must be supplemented by specific procedures in practice. In the capture rules framework, these procedures are the list of known capture rules and auxiliary procedures for inferring monotonicity constraints etc. [14]. In our framework these procedures include algorithm check clique, the two auxiliary procedures called by check clique, and other procedures for inferring monotonicity constraints and boundedness of predicates.
While we make no claims about the relative merits of the two frameworks, the results described in this paper allow us to detect the safety and effective computability of all queries that can be “captured” using the capture rules described in the literature [12, 10, 1, 14]. On the other hand, several of the examples presented here cannot be handled using the previously known capture rules. (Of course, it is possible to view the algorithm check_clique as a capture rule, if one relaxes the requirement [12] that the applicability of a capture rule be testable in polynomial time.)

In the context of rewriting algorithms (e.g., the Magic Sets algorithm), conceptually we first rewrite the program and then test it for safety and effective computability. The second part of this paper considers how this can be done efficiently by taking advantage of our knowledge of the transformations. The main problem is that the rewritten program often has a clique structure that combines several cliques of the original program. Thus, although the original program has relatively small cliques (as we expect), it is likely that the rewritten program has large cliques. Given the complexity of algorithm check_clique, the cost of directly analyzing the rewritten program could be prohibitive. We deal with this problem as follows. The adored program is an intermediate program produced by the Magic Sets transformation and has a clique structure that is similar to that of the given program. We show how the rewritten program can be analyzed by considering the cliques in the adored program and without directly considering the cliques in the rewritten program.

The rest of this paper is organized as follows. We present definitions in Section 2. In Section 3, we present an overview of our approach to safety analysis. In Section 4, we provide an introduction to the problem involved in testing recursive cliques for safety. In Section 5, we develop algorithm check_clique and prove its correctness. The rest of the paper deals with the analysis of programs generated using the Magic Sets transformation, and we present our results in Section 7. In order to keep this paper self-contained, we first describe sideways information passing and the Generalized Magic Sets transformation in Section 6. We present our conclusions in Section 8.

2. DEFINITIONS

In this section, we review a number of basic concepts from the literature and introduce additional ideas and definitions that are used extensively in the paper. A Horn clause or (rule) is a formula of the form \( p(t) : \text{\{} q_1(t_1), q_2(t_2), \ldots, q_m(t_m) \text{\}} \) where \( p(t) \) and the \( q_i(t) \)'s are atoms. An atom is a predicate name followed by a list of arguments, each of which is a term. A term is a constant, a variable, or a \( n \)-ary function symbol followed by \( n \) terms. A ground term is a term containing no variables. An example of a function symbol is the \texttt{cons} operator of Lisp and Prolog. The atom \( p(t) \) is called the head of the rule, and the rest of the rule is called its body. A Datalog program is a finite set of Horn clauses.

It is well known [8] that such a program may be viewed as a set of equations with a least fixpoint solution (which assigns a set of tuples to each relation); the solution can be found by repeatedly applying rules until no more new tuples can be generated. This process is called bottom-up evaluation and is complete, i.e., produces the least fixpoint. A rule is applied as follows: for all substitutions \( \theta \), if for \( i = 1 \cdots n \), \( q_i(t) \theta \) is in the rule current extension of \( q_i \), then we add \( p(t) \theta \) to the extension of \( p \). A query is a rule without a head.

We will assume here that a query is a single predicate occurrence (i.e., atom) and that all its arguments are variables. The set of answers to a query \( q(X) \) is the set of all facts \( q(c) \) in the least fixpoint.

We partition a program into a set of facts that are stored in the base predicates (the extensional database or EDB), and a set of rules defining derived predicates (the intensional database or IDB). A derived predicate is one that appears in the head of a rule with a non-empty body. Without loss of generality, we assume that the EDB and IDB are disjoint sets of predicates.

We only consider programs in which all arguments are variables or constants. Further, every rule defining an IDB predicate can contain only variables as arguments. Terms containing function symbols are not allowed in rules or facts. We “approximate” such terms by allowing infinite base relations: the approximation results in a loss of information that may prevent us from detecting the safety of some programs, but will not cause us to incorrectly declare a program to be safe. Rules containing constants can be similarly approximated by rules containing no constants: an occurrence of a constant, say 5, is replaced by a new variable, say \( X \), and an atom \( b(X) \), where \( b \) is an EDB relation (that intuitively contains just the constant 5, although this information is lost). We use the name Extended Datalog to denote this class of programs. The following example illustrates how a program with function symbols can be approximated as an extended datalog program.

Example 2.1. The following is a program that concatenates two lists:

\[
\text{concat}([X|Y], Z, [X|U]) :- \text{concat} (Y, Z, U) .
\]

\[
\text{concat}([ ], Z, Z) :- .
\]

1 This is not a limitation of the methods presented here. We deal with queries containing constants by first rewriting the program. The tests for safety and effective computability are performed on the rewritten program that is evaluated bottom-up. This issue is discussed further at the end of Section 5 and in Section 6.
We approximate it as follows in extended datalog:

\[
\text{concat}(V, Z, W) : \rightarrow \text{concat}(Y, Z, U), \quad h(X, Y, V), \\
\quad h(X, U, W).
\]

\[
\text{concat}(X, Z, Z) : \rightarrow b(X).
\]

The predicate \( h \) is an infinite base predicate corresponding to the function symbol \( "\cdot" \). Conceptually, we assign \( A \) "name" (a constant) to each list. The relation for \( h \) contains all triples \((\text{element}, \text{list} \, 1, \text{list} \, 2)\) such that \( \text{list} \, 2 \) is \( \text{(the constant that denotes the term [element|list 1]} \).

In translating a Horn clause program with function symbols into an Extended Datalog program, there is some loss of information. In order to retain more information, and thus improve the approximation, we allow a set of integrity constraints (IC) to be specified over the EDB predicates. The set of facts in the EDB predicates must satisfy these constraints. Thus a database is a triple \((\text{EDB}, \text{IDB}, \text{IC})\). The set of facts corresponding to a predicate, say \( p \), in the database is called the extension of \( p \).

The facts in the EDB are just rules with an empty right-hand side in which all arguments (of the head predicate) are constants. The EDB may contain predicates that have an infinite number of facts. As we saw earlier, these infinite predicates are used to model arithmetic operations and terms generated by function symbols. The domain of an Extended Datalog program is an infinite set with a partial order on it. The set of constants that appear in the EDB are drawn from this set.

We use the convention that infinite base predicates are denoted by \( f, g, h, \ldots \), finite base predicates by \( a, b, \ldots \), and derived predicates by \( p, q, \ldots \). Argument places are referred to by the predicate name subscripted by the place number.

For instance, \( p_i \) denotes the \( i \)th argument position of predicate \( p \). Also, we use sequences to denote sets; e.g., \( p_1, p_2 \) denotes the set \( \{ p_1, p_2 \} \). Variables are denoted by upper case letters and constants are denoted by numerals.

Programs may contain predicates that are empty for all EDB instances. The presence of such predicates unnecessarily complicates the safety analysis. Consider the following rule being the only rule defining \( p \) (i.e., no exit rule):

\[
p(X) : \rightarrow p(Y), \quad X = Y + 1.
\]

The presence of such a rule would lead us to infer that an infinite number of tuples for \( p \) could be generated by repeated applications of this rule. However, since this is the only rule defining \( p \), the extension of \( p \) is empty. While identifying all such predicates is in general undecidable, sufficient conditions can be used to detect and remove many rules defining such predicates.

We now consider the issue of safety.

**Definition.** A query is safe if for every instance of the EDB that satisfies all integrity constraints, the set of answers to the query is finite.

A query is unsafe if there is some instance of the EDB that satisfies the integrity constraints and is such that the query has an infinite set of answers. A predicate is said to be safe (resp. unsafe) if the query that asks for all tuples in this predicate is safe (resp. unsafe). The integrity constraints we consider in this paper are finiteness constraints and monotonicity constraints.

**Definition.** A finiteness constraint \((\text{FC})\) over a base predicate \( r \) is of the form \( X \rightarrow Y \), where \( X \) and \( Y \) are sets of argument positions. An instance of \( r \) satisfies this constraint if and only if the following property holds: For each tuple \( t \) in \( r \), the set of tuples \( \{ s[X] | s \in r \text{ and } s[X] \neq t[X] \} \) is finite.

**Example 2.3.** Consider the predicate \( f(X, Y) \). If we define \( f(X, Y) \) by \( Y = \ast \times X \), then we have the finiteness constraints \( f_1 \rightarrow f_2 \) and \( f_2 \rightarrow f_1 \). If we define \( f(X, Y) \) by \( Y < 0 \) and \( Y > 0 \), there is no finiteness constraint between the arguments of \( "\times" \). If we define \( h(X, Y) \) by \( X > 0 \) and \( Y = 0 \) or \( Y = 5 \), we have the finiteness constraint \( h_1 \rightarrow h_2 \).

Let \( r_i \) and \( r_j \) be argument positions of predicate \( r \).

**Definition.** A monotonicity constraint is a pair \( r > r_j \).

The constraint \( r > r_j \) holds in an instance of \( r \) if in every tuple the value in the \( i \)th argument is strictly greater than the value in the \( j \)th argument. We may also specify a monotonicity constraint \( r > c \) (resp. \( r < c \)) where \( c \) is a constant. This constraint holds in an instance of \( r \) if and only if the value in the \( i \)th argument is strictly greater (lesser than) than the constant \( c \).

The above definition assumes that the values are drawn from a domain with a partial order. Although it is possible to consider many different orders, we assume that there is a single order. (This is only for ease of exposition, and this issue is discussed further following Example 4.2.)

**Example 2.4.** Consider the predicate \( f(X, Y) \). If we define \( f(X, Y) \) by \( Y = \ast \times X \), then we have the finiteness constraints \( f_1 \rightarrow f_2 \) and \( f_2 \rightarrow f_1 \). If we define \( f(X, Y) \) by \( Y = \ast \times X, X > 0 \), \( Y > 0 \), we have the finiteness constraints \( f_1 \rightarrow f_2 \) and \( f_2 \rightarrow f_1 \), and the monotonicity constraint \( f_2 > f_1 \). If we define \( f(X, Y) \) by \( Y < 0 \) and \( Y > 0 \), there is no finiteness constraint between the arguments of \( "\times" \) but the monotonicity constraint \( f_2 > f_1 \) holds. If we define \( f(X, Y) \) by \( Y > 0 \) and \( Y = 0 \) or \( Y = 5 \), we have the finiteness constraint

\[2\] By \( t[X] \) we denote the tuple formed from \( t \) by deleting argument positions not in \( X \). Note that this definition is strictly weaker than the traditional definition of a functional dependency; any finiteness constraint hold trivially for all finite predicates (and in particular, for all finite EDB predicates).
$f_1 \rightarrow f_2$, but there is no monotonicity constraint relating $X$ and $Y$.

**Definition.** A predicate $p$ is defined using predicate $q$ if there is a rule containing $p$ in the head and $q$ in the body. We denote this as $p \leftarrow q$. We say that $p$ depends on $q$ if $p$ is defined using $q$ or $p$ depends on $r$ and $r$ depends on $q$. We denote this as $p^* \leftarrow q$. A predicate $p$ is a recursive predicate if $p^* \leftarrow p$. Two predicates $p$ and $q$ are mutually recursive if $p^* \leftarrow q$ and $q^* \leftarrow p$. A clique is a maximal set of mutually recursive predicates. We assume that each non-recursive predicate forms a singleton clique.

**Definition.** For each fact that belongs to a derived predicate, there exists a finite derivation tree, which describes how the fact is derived from base facts using rules of the program. Let $p(\bar{e})$ be a fact for the derived predicate $p$. All nodes in the tree for $p(\bar{e})$ are labeled with facts and those labeled with facts for IDB relations are also labeled by rules as follows. The tree has $p(\bar{e})$ labeling its root. Each internal node is labeled by a fact, and by a rule that generates this fact from the facts labeling its children. A leaf is either labeled with a base fact or with a derived fact corresponding to the head of a bodyless rule, this rule also labels it. A base fact, or a fact for a bodyless rule, constitutes a derivation tree of height one.

Note that a fact may have many associated derivation trees.

**Definition.** A path in a derivation tree is a sequence of adjacent nodes in the tree starting at the sequence node that is closest to a leaf and going towards the root.

In order to reason with monotonicity constraints, we need the notion of an argument mapping. Our definition is similar to the definitions presented in [1, 9].

**Definition.** For each rule $r$, we obtain an argument mapping between the head atom and each derived atom occurrence in the body. Let $p$ be the head and $q$ be an occurrence of a derived predicate in the body of a rule $r$. An argument mapping $(q, p, r)$ is a graph with the set of nodes being the argument positions of $p$ and $q$. In this graph, there is an undirected edge between two nodes if the same variable occurs in the corresponding argument positions. We draw an arc from one node, say $n_1$, to another, say $n_2$, if variables $X$ and $Y$ appear in the corresponding argument positions, and we can infer that $X \rightarrow Y$ from the monotonicity constraints in this rule.

Without loss of generality, we assume that each atom occurrence has been given a unique name. A simple way to do this would be to use the rule number and the position of the atom in the rule, together, as the atom name. When it is convenient, we show the arguments of the atoms as well in writing down an argument mapping. We note that the third component of an argument mapping is redundant, but we include it for ease of exposition.

Observe that such edge (or arcs) may connect nodes corresponding to argument positions of the same predicate occurrence. Also observe that some $(q, p, r)$ may induce a graph that is identical to the graph induced by some triple $(q_1, p_1, r_1)$.

**Definition.** Every argument mapping is also a composite mapping. In addition, argument mappings $(p, q, r_1)$ and $(q, m, r_2)$ can be composed to yield a composite mapping $(p, q, r_1)(q, m, r_2)$ by identifying nodes corresponding to the same position in $q$. In particular, the composite mapping $M = (p, q_1, r)(q_1, q_2, r_1) \ldots (q_n, p, r_n)$, $n \geq 1$, represents a cyclic composite argument mapping, and we complete the cycle by joining the nodes corresponding to the same argument positions in the two instances of $p$ with undirected edges. Further, if no symbol $q_i$ appears more than once, this is a simple cyclic mapping or just a simple cycle. If the mapping $M$ is such that $p$ only appears at the endpoints, we say that it is simple with respect to $p$ (even if other predicates appear more than once in $M$).

Given a database $DB = \{EDB, IDB, IC\}$, consider the set of all derivation trees for facts in $DB$. (Henceforth, when we refer to derivation trees, it is understood that we refer to a tree in this set.) We observe that every path (with more than one node) in a derivation tree induces a rule sequence (i.e., a sequence of rules) also induces a composite argument mapping. (In addition to the rule sequence, we must consider, at each node in the part, which atom in the rule body was expanded.) We thus often speak of the composite mapping corresponding to a path in a derivation tree. While there is clearly a unique composite mapping for a given path, the converse does not hold; several paths in derivation trees may correspond to the same composite mapping. We say that all such paths correspond to the composite mapping.

As an example, consider a cyclic (composite) argument mapping $M = (p, q_1, r_1), (q_1, q_2, r_1), \ldots, (q_n, p, r_n)$. Consider a given $p$-fact. Rule $r$ can be used ("applied") to generate a $q_1$-fact by using this $p$-fact to instantiate the $p$-atom in the body (corresponding to the argument mapping $(p, q_1, r_1)$). Note that, in general, other facts are necessary as well in order to instantiate the remaining body atoms of rule $r$. The generated $q_1$-fact can then be used to generate a $q_2$-fact by applying rule $r_1$. Proceeding similarly, we obtain a sequence of facts generated by applications of rules in the rule sequence $r, r_1, \ldots, r_n$. Such a sequence of facts forms a path corresponding to the argument mapping $M$. (This path is directed from bottom to top in the derivation tree.)

Note that we can associate a unique rule sequence (cycle) with each composite (cyclic) argument mapping. The converse is not true in general; if a rule body contains more than one occurrence of predicate, it is necessary to specify which occurrence is "expanded" by the next rule in the sequence in
order to construct an argument mapping. Where there is no confusion, we will sometimes refer to rule sequences/cycles and composite argument mappings interchangeably. This allows us to simply specify a composite argument mapping as a sequence of rules. A rule cycle corresponding to a simple cyclic mapping is called a simple rule cycle.

3. OVERVIEW OF THE SAFETY ANALYSIS

In addition to safety, we are also interested in whether the set of answers to the query is effectively computable with respect to a given evaluation strategy. For the purposes of the next definition, we leave the details of an evaluation strategy unspecified, only noting that a strategy could involve computing one or more intermediate (temporary) relations in order to compute the answer set.

**Definition.** A query is effectively computable with respect to a given evaluation strategy if the strategy computes all answers and terminates and every intermediate relation is of finite size.

Effective computability implies safety, but the converse is not true. That is, there might be no evaluation strategy with respect to which a given query is effectively computable, even though it is safe. In order to discuss effective computability, we define a stronger notion of safety.

**Definition.** A predicate \( p \) is strongly safe if every derived predicate \( q \) such that \( p \rightarrow^* q \) is safe. A query is strongly safe if the query predicate is strongly safe. A program is strongly safe if every predicate in it is strongly safe. A clique is strongly safe if every predicate in it is strongly safe; A rule is strongly safe if every predicate that appears in its body is strongly safe.

We now present an overview of our approach to testing a program. Let \( C_1 \) and \( C_2 \) be two distinct cliques. Clique \( C_1 \) is a child (resp. descendant), of clique \( C_2 \) if \( p \rightarrow q \) (resp. \( \rightarrow^* \)) for some \( p \) in \( C_2 \) and \( q \) in \( C_1 \). We denote this as \( C_2 \prec C_1 \) (resp. \( C_2 \prec^* C_1 \)). We observe that \( \rightarrow^* \) is transitive and anti-symmetric. A clique \( C \) is a leaf clique if there is no clique \( C \) such that \( C \prec C_1 \). We test program \( P \) for strong safety according to this ordering of cliques. We begin with cliques that are leaves. All predicates in such a clique are either base predicates or predicates that are defined in it.

Let \( C_1 \) be a leaf clique, and let \( q \) be a predicate in \( C_1 \) that is used to define some predicate in another clique \( C_2 \). After analyzing clique \( C_1 \), if it is strongly safe, we replace all occurrences of \( q \) by a base predicate (If we cannot show this clique to be strongly safe, then we cannot show the program—which contains predicates from this clique—to be strongly safe.) After doing this for all predicates in \( C_1 \) that are used to define predicates outside \( C_1 \), we remove \( C_1 \) from the set of cliques. We then consider another leaf clique and proceed in this way until all cliques have been considered.

In order to reason about effective computability, we must first specify the model of execution.

In this paper, the model of execution is that a program is evaluated bottom-up by repeatedly applying the rules (in any fair order) until no new tuples are produced by the application of any rule.\(^4\) A rule is applied by taking the join of (the current extensions of) the relations in the body of the rule and projecting out tuples corresponding to the head.

We assume a left-to-right order in computing the join of the body relations.

We need a notion of finitely evaluable rules in the context of infinite base relations and finiteness constraints. To this end, we make the following important assumptions about programs:

1. Every variable in the head of a rule also appears in the body.
2. If an infinite base relation \( f \) appears in the body of a rule, \( \{f_1, ..., f_n\} \) be the set of argument positions such that the variables in these positions appear to the left to this occurrence of \( f \) in the body of the rule. Then, these arguments must \( \rightarrow^* \) determine all arguments of \( f \) through the FCs given to hold over \( f \).
3. Consider an infinite base relation \( f \) with an FC \( f_{i_1}, ..., f_{i_n} \rightarrow f_k \). Given an assignment of domain values to argument positions \( f_{i_1}, ..., f_{i_n} \), there is given procedure that “when called” returns the (finite) set of values appearing in the argument position \( f_k \).

The first assumption is made primarily for ease of exposition. It allows us to ignore single rule applications that can generate an infinite set of values. (Note that the presence of such rules defining some IDB predicate does not necessarily imply that the query predicate is unsafe.) The second and third assumptions summarize the way in which we utilize FCs. Informally, we can think of a rule being unified with known facts in a left-to-right order; that is, we first unify the leftmost body atom with a fact and then proceed to the next atom and so on. When we encounter an infinite EDB atom, we must find a fact in the infinite EDB relation that can unify with the (partially instantiated) atom. Clearly, we do not expect to store all facts in an infinite EDB relation; rather, there is some procedure that will compute matching facts given domain values for a sufficient number of argument positions. For example, consider \( \text{plus}(X, Y, Z) \), with the meaning of \( X + Y = Z \). Given values for \( X \) and \( Y \), we can compute the unique \( Z \) value, and thus the (only) matching \( \text{plus} \) fact for the given values of \( X \) and \( Z \). Assumption 2

\(^4\) We expect that a given program \( P \) is first rewritten, using the Magic Sets transformation, to \( P' \), which is then evaluated bottom-up. In later sections, we discuss how the structure of these rewritten programs can be utilized to make the safety analysis more efficient.
states that the set of matching facts for an infinite EDB atom (when we encounter it in a left-to-right instantiation of the rule) is finite, and Assumption 3 states that we must have a procedure to compute this set of facts. (In two stages, first a finite superset of candidate tuples is constructed; then, these are checked one by one.)

We conclude this section by formally presenting some important properties of this model of execution. In particular, Theorem 3.2 characterizes the effective computability of strongly safe programs.

We define an application of a rule to be effectively computable if it is possible to compute the join of the body predicates in left-to-right order without constructing infinite intermediate results, and the set of tuples thus produced for the head predicate is also finite.

**Lemma 3.1.** Let a program satisfy the above three assumptions. Further, let the initial extension of each derived predicate be finite and consider a computation (in our model of execution). Then, the $ith$ rule application, for positive integer $i$, produces only a finite number of tuples for the head predicate and is effectively computable.

**Proof.** The proof is by induction on $i$, the rule application number. The basis is the very first rule application, $i = 1$.

A rule application consists of taking the join of the body predicates in left to right order. The only potential difficulty arises when the next predicate to be joined is an infinite base relation (since the current relation for each derived predicate is finite initially). The left to right evaluation provides a vector of bindings for a non-empty subset of the arguments (let us call this subset the bound arguments) of the infinite relation, and by Assumption 2 these arguments determine values for all the other arguments. Given a vector of values for the bound arguments, the corresponding vectors of values for the other arguments are determined using the procedure of Assumption 3. This gives us a finite relation for this step of the join. Further, every variable in the head appears in the body. This ensures that the set of tuples produced for the head is also finite and completes our proof of the basis case.

For the induction step, let the claim hold for the first $i − 1$ rule applications. This ensures that the current relation is finite for all derived predicates when the $ith$ rule application is carried out, since such relations are the unions of the finite sets of tuples produced in the preceding $i − 1$ rule applications. With this observation, the proof for the basis case applies to the inductive case as well, concluding our proof of the lemma.

We now consider the notion of effective computability in the context of the left-to-right bottom-up model of execution presented above. A query was defined to be effectively computable if its evaluation terminated after computing all answers, and only finite intermediate relations were constructed. For the bottom-up model of execution, the intermediate relations are of two kinds—the derived predicates in the program and the temporary relations created in the application of a rule (to evaluate the joins of body predicates); Since the extensions of derived predicates are initially empty, applying Lemma 3.1 inductively, we know that the temporary relations created in any rule application are finite. Thus, to show a query to be effectively computable in our model of execution, we need to show that every derived predicate is safe and that the computation terminates after a finite number of rule applications. We must also show that termination of tuple generation can be detected.

We have the following theorem characterizing strongly safe queries.

**Theorem 3.2.** Under the assumptions of Lemma 3.1, if a query is strongly safe, then it is effectively computable in the above model of execution.

**Proof.** First, we limit the program by eliminating all rules whose head predicates do not, directly or indirectly, define the query. By Lemma 3.1, each rule application is effectively computable, given the three assumptions about programs. Thus, we only need to show that initially extensions for all derived predicates are finite (which is true as these extensions are initially empty in our model of computation) and that the computation halts after a finite number of rule applications.

For every predicate $q$ that is used to define the query, strong safety implies that $q$ is safe (i.e. finite) and so, by the completeness of bottom-up evaluation, all tuples in it will eventually be produced. When this has been done for all predicates, further rule applications will not produce any new tuples, and the computation will terminate. (recall that this is the standard termination condition for a bottom-up fixpoint computation.)

To summarize, our approach is to test for strong safety on a clique by clique basis. Theorem 3.2 then ensures effective computability according to our model of execution given our three assumptions about programs.

### 4. An Introduction to Testing Recursive Cliques for Strong Safety

We now introduce some technical definitions used in the safety analysis and through illustrative examples on recursive cliques containing a single rule, we outline the safety check for such simple recursions.

**Definition.** A property over domain $D$ is a predicate, or condition, over sequences of elements from $D$. 
DEFINITION. A property over domain $D$ is well-founded if the following holds: (1) There is a total order $\geq$ on $D$ such that for all $c_1, c_2$ in $D$, if $c_1 \geq c_2$, there are only a finite number of values $c$ in $D$ such that $c_1 \geq c \geq c_2$. (2) Let $S = e_1, e_2, \ldots$ be an arbitrary sequence that satisfies the property. For all adjacent elements $e_i, e_{i+1}$ in the sequence $S, e_{i+1} \geq e_i$, and (3) there is an element $c$ in $D$ such that $c \geq e_i$ for all $e_i$ in $S$.

As an example, consider the following property: All sequence elements are integers greater than 10 and the sequence is increasing. This property is not well founded since we cannot satisfy condition (3). However, suppose that the definition of the property additionally requires that all elements should be less than 100, then it is well founded. Taking the ordering $\geq$ to be the greater-than relation $>$ over the integers, in any sequence satisfying the property, the element 100 serves as $c$ in condition (3). As another example of a well-founded property, consider the following: All sequence elements are greater than 10, and the sequence is decreasing. Here, $>$ is the less-than relation, and $c$ is 10.

DEFINITION. Consider a property $p$ and a predicate $p$. Consider a path in a derivation tree. We say that this path satisfies property $p$ over predicate $p$ if the sequence of facts of the form $p(\ )$ on this path satisfies property $p$. We also say that the path satisfies ($p$, $p$), for brevity.

We will identify well-founded properties as a means of arguing that a sequence of rules cannot be applied repeatedly to produce an infinite set of tuples. A large class of well-founded properties over sequences of facts can be specified by simply specifying an argument position, a domain of values that can appear in this position, an ordering, and a bound ($c$). In fact, all examples in this paper (with the exception of Example 4.2) use only a subset of this class, with a single ordering of values. The consideration of more sophisticated orderings, possibly over combinations of arguments, or the use of several different orderings in showing safety of a single program, presents no difficulties.

A cycle in an argument mapping is a cycle in the corresponding graph when directions of arcs are ignored. An increasing (resp. decreasing) cycle in the argument mapping is a cycle such that the following holds: In any derivation tree, in every path corresponding to the argument mapping, such that rule $m$ labels both the lowest node in the path and the node, on this path, which is closest to the root of the tree, the values in argument positions that correspond to the nodes on the cycle in the argument mapping are increasing (resp. decreasing) with regard to the total order that is defined on the database domain. (Recall that paths are directed towards the root.)

We say that a variable $X$ in a rule is bounded if it can only take on values from a finite domain. An argument position in an atom occurrence is bounded if the variable in it is bounded. Similarly, we say that variable $X$ is bounded above (resp. below) if it can only take on values from a domain that is bounded above (resp. below). A cycle in an argument mapping is bounded above (below) if it contains at least one node (i.e., an argument position) that is bounded above (below).

Note that every path corresponding to an increasing (resp. decreasing) cycle satisfies a simple increasing (resp. decreasing) property. If, additionally, some argument position on the cycle is bounded above (resp. below), then the property is well-founded. The (resp. increasing or decreasing) property serves as $\geq$, and the (resp. maximum or minimum) values in the finite set of values that the argument position can take serves as $c$ in the definition of a well-founded property. If we can associate a well-founded property with a cycle in an argument mapping, this ensures that paths corresponding to this cycle can only be repeated a finite number of times in any path in a derivation tree. Thus, applications of this cycle of rules can only produce a finite number of new facts. Consequently, any recursive single rule clique is safe if a well-founded property is associated with the (exactly one) cyclic argument mapping in the clique. These observations are made precise in Section 5, but we illustrate the ideas in the following example.

EXAMPLE 4.1. Consider the following program:

1. $p(X,Y) := p(U,V), X = U - 1, Y = V + 1, b(X)$.
2. $q(X,Y) := q(U,V), X = U - 1, Y = V + 1, b(X)$.
3. $p(X,Y) := c(X,Y)$.
4. $q(X,Y) := d(X,Y)$.

There are two recursive cliques, say $C_1$ and $C_2$, containing $p$ and $q$, respectively. Consider the composite argument mapping in $C_1$: $(p(U,V), p(X,Y), 1)$. There is an arc from $U$ to $X$, and one from $X$ to $V$. Thus, we have an increasing cycle and a decreasing cycle in this argument mapping. Further, $X$ is bounded since it appears in the finite base relation $b$, and so the first cycle of rules can be applied only a finite number of times. Each rule application can only produce a finite number of values. The clique $C_1$ is therefore strongly safe. A similar analysis shows $C_2$ to be strongly safe as well. Now, let us add the following rule:

5. $p(X,Y) := q(Y,X)$.

This changes the clique structure by making $C_1$ depend on $C_2$. However, $C_1$ and $C_2$ are still the only cliques, and the previous analysis holds. Indeed, $q$ is strongly safe, and the tuples it contributes to $p$ may be thought of as being in a finite base relation. We now add a sixth rule to the program:

6. $q(X,U) := p(U,V), b(X)$.

This changes the clique structure drastically. The two cliques are merged into a single clique containing both $p$.
and \( q \). In addition to the two cyclic mappings considered before, there is the cyclic mapping \((q(Y,X), p(X,Y), 5) (p(U,V), q(X,U), 6)\). Intuitively, rule cycle \((2,2)\) can be used to produce a value (on which there is no bound) in the second argument position of \( q \); subsequently, rules 5 and 6 are applied to generate a \( q \)-tuple that has this value in the second argument position. The rule cycle \((2,2)\) can be applied again to increase the value in the second position. By repeating this sequence of rule applications, an infinite number of tuples can be generated.

Consider a vector of \( m \) arguments. It is possible to identify properties that characterize the vector (or some subvector) rather than individual arguments. Examples of such properties include the sum of the arguments (assuming that they are numerical arguments), and the interpretation of the argument vector as a character string. We can also define orderings with respect to such properties. The sums of arguments can be ordered by arithmetic “\(<\)”, and argument vectors viewed as character strings can be ordered using the lexicographic ordering over character strings.

**Example 4.2.** We illustrate the added power of considering properties over several arguments.

1. \( p(X,Y) := p(U,V), X = U - 2, Y = V + 1, X + Y > 0, 2. \( p(X,Y) := c(X,Y).\)

\( c \) is a finite base relation. This program is safe since the sum of \( p \)'s arguments is monotonically decreasing and is bounded below by 0. It is not clear how to establish a bound on either argument position by considering it separately.

We note that while we have so far considered monotonicity constraints with respect to a single ordering over the domain, it is easy to extend our definitions when several different orderings are used. (In effect, arcs in argument mappings would be subscripted by the ordering used to define them and, similarly, notions of increasing and decreasing cycles and bounded nodes would be with respect to given orderings.) It is also possible to define orderings over cross-products of the domain. This is necessary when we wish to specify some well-founded property over a set of arguments rather than a single argument, as in the previous example. (The definition of an argument mapping in this case would include arcs from sets of arguments to sets of arguments.) Similarly, we could define orderings over cross-products of the domain if we wished to specify some well-founded property over argument vectors. These are straightforward extensions and allow us to define a variety of well-founded properties over arguments or sets of arguments of a predicate.

The purpose of this section has been to give the reader some intuition about the problem of testing recursive cliques for safety and effective computability. In the next section, we develop these ideas rigorously.

### 5. Testing Arbitrary Cliques for Strong Safety

In the previous section we considered how a clique containing a single recursive rule cycle could be tested for strong safety. If there are two or more such cycles involving the predicates in a clique, the problem becomes more complex because these rule cycles could interact in such a way that the tuples produced by one of them invalidate the assumptions made in declaring one of the others strongly safe. (We encountered this problem when we added rule 6 to Example 4.1.)

Consider the argument mappings corresponding to two rule cycles:

1. The monotonic (increasing or decreasing) cycles considered in the two mappings may involve different argument positions. Thus, the application of one of these rule cycles may introduce arbitrary values into argument positions not on the monotonic cycle in its own argument mapping (but possibly on the monotonic cycle used to check one of the other rule cycles, thus invalidating that check).
2. The monotonic cycles considered in the two mappings may involve different orderings. Thus, although the same argument positions may be involved, a rule cycle may introduce arbitrary values into these positions with respect to any ordering other than used in checking it.

**Example 5.1.** The following example illustrates the second point above:

1. \( p(X,Y) := p(U,V), X = U - 1, Y = V + 1, b(X).\)
2. \( p(X,Y) := p(U,V), X = U + 1, Y = V + 1, b(X).\)
3. \( p(X,Y) := c(X,Y).\)
4. \( c(1,11).\)
5. \( c(2,12).\)
6. \( b(1).\)
7. \( b(2).\)

The two simple rule cycles are \((1,1)\) and \((2,2)\). The first argument is part of a decreasing cycle in rule cycle \((1,1)\) and part of an increasing cycle in rule cycle \((2,2)\). In both, the first argument is bounded because \( X \) occurs in the finite base relation \( b \). So each of these rule cycles is safe if it is the only rule cycle. However, taken together, they are unsafe. It is possible to alternate between the two cycles so that the first argument alternates between the values 1 and 2 (which are the only values in \( b \)), and on each application the second argument increases by 1. Thus, the second argument can be any integer greater than 10 (since the least value in \( c \) is 11).

In order to deal with arbitrary cliques, we must analyze the interactions between the simple rule cycles generated by
the rules in the clique. The following theorem underlies our approach.

**Theorem 5.1.** Consider a database $DB = (EDB, IDB, IC)$. If for all EDBs there exists a constant $h$ such that the height of every derivation tree is less than $h$, then every predicate is strongly safe.

**Proof.** By Lemma 3.1, each application of a rule produces only a finite number of facts. If the heights of all derivation trees are bounded, then there is a maximum number of rule applications involved in producing any fact. Thus, the number of possible facts that can be produced by rule applications is also finite.

We now present an algorithm to test for safety and prove its correctness by establishing that if the algorithm certifies a program, then all derivation trees are of bounded height (Theorem 5.1). The details of the algorithm are as follows:

First, we identify all the simple rule cycles in the clique and generate the corresponding (cyclic) argument mappings. Then we test each of these cyclic argument mappings $((p, \ldots), (\ldots, p, \ldots))$ as if it were the only cyclic mapping in the clique. In doing this, we check that there is some property defined over the arguments of $p$, say $\text{prop}_p$, that is well-founded with respect to this mapping. Having done this for all simple cyclic composite argument mappings, we verify that their interaction does not invalidate the assumptions made in checking them individually. For example, if $((p, \ldots), (\ldots, p, \ldots))$ is considered well behaved because the first argument decreases on each application of this cycle of rules and is bounded below, we must check that applications of every other rule cycle either leaves the value of the first argument unchanged or decreases it.

We make use of two procedures called $\text{check}_\text{prop}$ and $\text{tag}_\text{prop}$. We now specify them formally:

* boolean $\text{check}_\text{prop}(C, \text{prop})$. Consider a cyclic argument mapping $C = (p, q_1, r), (q_1, q_2, r_1), \ldots, (q_n, p, r_n)$. The procedure $\text{check}_\text{prop}(C, \text{prop})$ returns true if the following holds: Every path $P = (p( ), q_1( ), q_2( ), \ldots, q_n( ), p( ))$ in a derivation tree satisfies $\text{prop}_p$.

* $\text{tag}_\text{prop}(C, \text{prop}, q, \text{prop})$. Consider a cyclic mapping $C = (p, q_1, r), (q_1, q_2, r_2), \ldots, (q_n, p, r_n)$. For $j = 1$ to $n$, the procedure $\text{tag}_\text{prop}(C, \text{prop}, q, \text{prop})$ returns $\text{PROP} = \text{prop}_p$. The returned properties $\text{prop}_1, \text{prop}_2, \ldots, \text{prop}_n$ are some $n$ properties that satisfy the following:

  - Let $Qi = (q_i( ), \ldots, q_i( ), i = 1$ to $n$, be any $n$ paths such that $Qi$ satisfies $\text{prop}_i$. Then $P = (p( ), q_1( ), q_2( ), \ldots, q_n( ), p( ))$ be any path in a derivation tree that satisfies $\text{PROP}$. Then the path $(p( ), q_1( ), \ldots, q_{i-1}( ), q_i + 1( ), \ldots, q_n( ), p( ))$ also satisfies $\text{PROP}$.

  - If for some $i$, $\text{prop}_i$ satisfying the above either cannot be found or simply does not exist, $\text{PROP} = \text{false}$ is returned.

Intuitively, $\text{tag}_\text{prop}$ accepts a cyclic argument mapping $C$, a property $\text{prop}$ and a node $qi$ on cycle $C$, and it identifies a property $\text{PROP}$, such that the following is true of all paths $P$ and $Qi$. Let path $P = (p( ), q_1( ), q_2( ), \ldots, q_n( ), p( ))$ satisfy $\text{PROP}$ and path $Qi = (q_i( ), \ldots, q_i( ))$ satisfy property $\text{prop}_i$. (Path $P$ corresponds to a sequence of rule applications $r_1, r_2, \ldots, r_n$) Then the path that is obtained by replacing node $qi( )$ in $P$ by the path $Qi$ also satisfies property $\text{PROP}$.

For example, if $\text{PROP}$ is the property that some argument of $p$ is increasing then there is some increasing cycle in the argument mapping $C$ that contains occurrences of this argument of $p$. If some argument of $qi$ appeared in this cycle, then $\text{PROP}$ is the property that this argument of $qi$ should increase or remain the same. (Note that $\text{PROP}$, in this example is not necessarily a well-founded property.)

**Remark.** We shall refer from now on to argument mappings using the triple-sequence notation. All details of examining cycles are abstracted by using procedures $\text{check}_\text{prop}$ and $\text{tag}_\text{prop}$.

We present a recursive procedure $\text{check}_\text{clique}$ that examines the interactions between rule cycles. We prove that if this procedure returns $\text{true}$, then all paths in a derivation tree built using rules from the given clique are bounded in length, where the bound is a function of the EDB. Thus, the number of derivation trees for facts in EDB is bounded. (This implies that every clique predicate is safe, and it immediately follows that every predicate is strongly safe as well. Effective computability then follows from Theorem 3.2.)

The intuition behind the algorithm is as follows. Suppose that each simple rule cycle, by itself, preserves some well-founded property which ensures that this cycle can only be repeated a finite number of times along a path in a derivation tree. However, a predicate $q$ that appears on this path can be “expanded” by a path corresponding to some composite argument mapping of the form $(q, \ldots, r, \ldots, q, \ldots, q, \ldots)$. The expanded path may not preserve the well-founded property.

Our basic observation is that any path can be decomposed hierarchically. (We present the idea briefly, and discuss it in detail immediately before Lemma 5.3, where it plays a central role.)

**Definition.** A path segment is a path in a derivation tree of the form $p( ), \ldots, p( )$, for some predicate $p$. The segment is simple with respect to $p$ if $p( )$ does not appear in it except for the endpoints.

To recognize the hierarchical structure, we proceed as follows. First, all consecutive occurrences of some predicate $p$ are marked. Consider each simple path segment (including endpoints) w.r.t. $p$ that is obtained, and within it mark all consecutive occurrences of the first predicate $q$, that is
We consider the hierarchy from the leaves to the top of the hierarchy, we have path segments with $p$-facts at the endpoints, and these segments must preserve the property associated with predicate $p$. At each level of the hierarchy, the property that must be preserved is the property associated (by procedure $tag_{prop}$) with the predicate used for marking the segments at this level. At the leaves of the hierarchy, we have paths that correspond to simple cycles of argument mappings. We prove that at each level the associated properties are indeed preserved, using induction on the level number. The basis is established readily by considering calls to $check_{prop}$ for the simple cycles at the leaves of the hierarchy. The induction relies upon calls to $tag_{prop}$.

We note that a given rule cycle may be used to expand any predicate that occurs on it, and so if the cycle contains two predicates $p$ and $q$, the corresponding argument mapping can be denoted as either $(p,\ldots,\ldots,p)$ or $(q,\ldots,\ldots,q)$. The fact that they refer to the same rule cycle should be recognized and utilized in testing whether these mappings satisfy some property, for the purposes of the subsequent analysis it is convenient to assume that both representations are generated. Thus, by referring to all simple cyclic argument mappings of the form $(p,\ldots,\ldots,p)$ or $(q,\ldots,\ldots,q)$ we are assured that indeed all simple rule cycles containing $p$ are considered.

The following procedure checks an arbitrary clique for strong safety. We assume that all predicates are either members of the clique or defined non-recursively, and we also ignore some optimizations for the sake of clarity.

**proc check_clique** $(C)$. boolean /* The top-level procedure */

1. For each predicate $pi$ defined in the given clique $C$, we associate well-founded property $prop_i$.
2. $FLAG := true; i := 1$
   while $FLAG$ and $i \leq n$ do /* $n$ is the number of predicates defined in $C$ */
   $FLAG := verify(pi, \{\}, prop_i);$  
   $i := i + 1$
   od.
3. Return $FLAG$.
end /* check_clique */

**proc verify** $(p, IGNORE, rop)$. boolean

/* Does the detailed work. Checks if $prop$ is a well-founded property for each simple cyclic composite argument mapping $(p,\ldots,\ldots,p)$. To do this, it must check for interactions with other simple argument mappings. $IGNORE$ is a set of predicates. In testing for interactions, we ignore all argument mappings in which some predicate in $IGNORE$ appears at some position other than the endpoints. This is required for the algorithm to terminate, and is discussed later.*/

/* Assumes that two procedures, $check_{prop}$ and $tag_{prop}$ are given.*/

/* Let $\{C1, \ldots, Cm\}$ be the set of simple argument mappings of the form $(p,\ldots,\ldots,p)$, and let, for $i = 1$ to $m$, $Ci'$ be the set of predicates appearing on $Ci = (p,\ldots,\ldots,p)$, except for $p$.*/

1. /* First verify that $prop$ is a well-founded property for each of the cycles $Ci$ considered separately.*/

   $FLAG := true;$
   for all $i$ in $\{1,\ldots,m\}$
   if $Ci \cap IGNORE = \{\}$ then /* do not consider cycles containing predicates in $IGNORE$ */
   /* $check_{prop}$ verifies that $(p,\ldots,\ldots,p)$ is preserved by the sequence of predicates on $Ci$ if they are not expanded further */
   $FLAG := FLAG \& check_{prop}(Ci, prop);$  
   od.
2. /* Now, for each cycle $Ci$, we ensure that the other cycles preserve the well-foundedness of $prop$. (That is, when these cycles are used to expand predicates on cycles for $p$).*/

   for all $i$ in $\{1,\ldots,m\}$
   if $Ci \cap IGNORE = \{\}$ then
   Let $q1,\ldots,qk$ be the recursive predicates in $Ci'$.
   for all $j$ in $\{1,\ldots,k\}$
   $tag_{prop}(Ci, prop, qj, prop_i);$  
   $IGNOREQ := IGNORE \cup \{p\};$
   $FLAG := FLAG \& verify(qj, IGNOREQ, prop_i);$  
   od.
3. return $(FLAG)$.
end /* verify */

We now present the main theorem.

**Theorem 5.2.** Let $C$ be a clique such that all derived predicates used to define predicates in $C$ are either in $C$ or are defined outside $C$ and denote finite relations. If $check_{clique}$ $(C)$ returns true, then there is a constant $L_{(C,DB)}$ such that the longest path in any derivation tree is less than or equal to $L_{(C,DB)}$.

**Proof.** If $check_{clique}$ returns true, we prove that there is a bound $L$ such that all paths from a leaf to an internal node in a derivation tree are at most of length $L$. (In this proof, by “path,” we always refer to a path from a leaf to an
internal node.) We argue by induction on the number $k$ of recursive predicates that appear on a path. Let us define $\text{RSym}(\text{path})$ to be the set of recursive predicate symbols $p$ such that, for some $c$, there is a fact $p(c)$ on the path.

**Induction hypothesis.** There exists a constant $L_{(C, DB)}$ such that for every path from a leaf to an internal node, if $|\text{RSym}(\text{path})| < k$, then the length of the path is less than $L_{(C, DB)}$. (We use $|S|$ to denote the cardinality of set $S$.)

**Basis.** Consider a path with only one recursive predicate, say $p$. Let the path be $q(1), ..., qn(1), p(1), ..., p(l)$, where the $q_i$ are non-recursive predicates. Since there is only one recursive predicate, the segment $p(1), ..., p(l)$ must correspond to some applications of simple argument mappings of the form $(p, ..., q)$, $(q, ..., p)$. Procedure check_clique ensures that each application of such an argument mapping preserves the well-founded property associated with $p$. (This was done through the call to verify in check_clique, and the call to check_prop in Step 1 of this call to verify.) Thus, for any two occurrences of $p$ on this path such that there are no occurrences of $p$ in between (henceforth, we refer to such occurrences as adjacent occurrences), the value of the occurrences uniformly increases (or decreases).

Let $p(c)$ be the first occurrence of $p$ in the path. The vector of values $c$ is obtained from values in the database by one or more applications of non-recursive predicates: $q(1), ..., qn(1)$. From Lemma 3.1, it follows that the set of possible values $c$ is finite. Given the well-founded property associated with the segment $p(1), ..., p(l)$, this ensures that the length of the path is bounded by some constant $L_{(C, DB)}$ (which depends on the clique $C$ and values in the database $DB$).

**Induction step.** Consider a path in an arbitrary derivation tree. We have to show that it is bounded in length by some constant depending on $C$ and $DB$. Furthermore, this path has $k$ different recursive predicates in it. Let $p$ be the recursive predicate occurring most often on this path. So the path can be written as

$$r1(1), ..., ry(1), q1(1), ..., qy(1), p(1), ..., p(l), u1(1), ..., um(1)$$

where $r1(1), ..., ry(1)$ are non-recursive atoms and both $q1(1), ..., qy(1)$ and $u1(1), ..., um(1)$ are recursive atoms but do not include $p$. The segment $p(1), ..., p(l)$ may contain occurrences of $p$ and other recursive predicates. The segment $r1(1), ..., ry(1), q1(1), ..., qy(1)$ has occurrences of less than $k$ recursive predicates. So, its length, by three induction hypothesis, is bounded by some constant $\text{CONST}$ that depends only on the DB and $C$. Consider the set $W$ of all the values that can be produced by $\text{CONST}$ rule applications starting from a finite DB: By our assumptions about programs, there are finitely many values in $W$. Let $d$ be the smallest value according to $\gg$ in $W$: such a least value exists since $\gg$ is a total order. The arguments in the first occurrence of $p$ are $d$ (or equal to $d$). Now look at the segment $p(1), ..., p(l)$. The well-founded property is preserved between two consecutive occurrences of $p$, as we shall prove in Lemma 3.3. Let $c$ be the bound of the well-founded property and let $d + c$ denote the number of elements in $D$ “between” $d$ and $c$; this number is finite by definition of a well-founded property. It follows that there cannot be more than $d + c$ occurrences of $p$ in $p(1), ..., p(l)$. Now, since $p$ is the recursive predicate that appears the most on the path $r1(1), ..., um(1)$, all other recursive predicates appear at most $d + c$ times. Since only recursive predicates can appear on the segment $p(1), ..., p(l)$, $u1(1), ..., um(1)$, and the number of recursive predicate symbols is determined by $C$, it follows that the length of this segment is at most $\text{CONST} = (d + c) |C|$. Thus, the length of the path is bounded by $\text{CONST} + \text{CONST}$.

Since we treated an arbitrary path, all path lengths in derivation trees are bounded by a constant depending on the database and $C$. 

The following lemma is central to the proof. The main technical difficulty is in establishing a formal correspondence between the recursive calls in check_clique and paths in derivation trees. We use a transformation $T$ to achieve this. We first prove that every cyclic path in a derivation tree can be produced by applications of this transformation (Lemma 5.4). We then prove that every path $p(1), ..., p(l)$ so produced preserves $(p, \text{prop})$, where $\text{prop}$ is the property associated with $p$ (Lemma 5.5). To prove the latter claim, we define a structure called a transformation tree that reflects the sequence of applications of $T$ used to produce a given path and establish a correspondence between this structure and the sequence of recursive calls in check_clique.

We first introduce a transformation $T$ that operates on argument mappings (corresponding to segments of paths in derivation trees). The transformation $T$ takes an argument mapping $M = \langle q1, q2, r1, ..., (p, qi, r), ..., (qn, qr, rm) \rangle$ as input. The output is an argument mapping that is identical to $M$ with the exception that the mapping $(p, qi, r)$ is prefixed with some cyclic argument mapping of the form $(p, ..., r1, ..., (p, qi, r))$ that is simple with respect to $p$. It is often convenient to think of $T$ as operating on path segments in a derivation tree—essentially, it “expands” a node of the form $p(1)$ in the input path by a cyclic path of the form $p(1), ..., p(l)$ of course $T$ operates on the argument mappings corresponding to these paths; this is to be understood although we sometimes abuse notation and refer to the path segments (corresponding to argument mappings) produced by $T$.

Given an arbitrary path $P = p(1), ..., p(l)$ in a derivation tree, consider a sequence of applications of $T$ that generates...
the (composite) argument mapping corresponding to this path. The sequence of applications of $T$ can be visualized as a tree rooted at the node $p$. Initially, the tree consists of just the root, and each application of $T$ adds a leaf node in the current tree. The sequence of leaves produced by one or more applications of $T$ is always an argument mapping of the form $(p, ..., p)$. The transformation tree is essentially this structure, with each node additionally annotated by a superscript that contains some information that makes it easier to establish a correspondence with calls in check_clique; we now define transformation trees formally.

**Definition.** A transformation tree rooted at $p( )$ is a tree that is induced by a sequence of steps of transformation $T$ starting from a node $p$( ). It is defined as follows:

1. For the empty sequence of steps, it is the node $p( )$, where $Pr$ is the property associated with predicate $p$.

2. Let $R$ be the transformation tree associated with the first $k$ steps of the sequence of steps of $T$. Let Step $k+1$ expand a leaf $q( )$ with a simple cyclic argument mapping $C_q$ that corresponds to a simple path $(q(), ... , q(), ... , q())$. The transformation tree $S$ associated with the first $k+1$ steps of the sequence is obtained from $R$ by making $(q(), ... , q(), ... , q()) = (q', ... , q', ... , q')$, in that order, the children of $(q())$ in $R$ $q( )$, where $I_q = I_q \cup \{q\}$, and $Prq_i$ is obtained using the procedure call tag_prop($C_q, Prq, q, Prq_i$). (Note that we can easily recover the argument mapping $C_q$ from the sequence of children of the expanded node.)

Thus, the sequence of leaves denotes the argument mapping that is the input (and subsequently, the output) for each application of $T$. We place the following restriction on applications of $T$ in the construction of a transformation tree:

Consider the leaf, say $q( )$, chosen for expansion. The simple cycle used to replace $q( )$ should not contain any of the predicates in $I_q$.

We will only consider applications of $T$ in which the above restriction is satisfied; this is to be understood whenever we refer to applications of $T$ in the rest of this paper. Further, it is often the case that the entire tree is not of interest, and we will refer to the sequence of leaves of the transformation tree, read from left to right, as the result of the applications of $T$.

The role of the set IGNORE in check_clique is closely related to the restriction on applications of $T$. In essence, we introduced the set IGNORE to reduce the number of argument mappings that we examine in check_clique; this reduction is necessary to ensure termination of the algorithm. The restriction on $T$ similarly restricts applications of $T$. There is a direct correspondence between applications of $T$ and calls to verify. The set IGNORE in a call is identical to the first component of the superscript of the expanded node in the corresponding application of $T$. This correspondence is utilized in Lemma 5.5.

**Lemma 5.3.** Let $p$ be a recursive predicate. If check_clique $C$ returns true, then every segment between two occurrences of $p$ in a path of a derivation tree preserves the well-founded property associated with $p$.

**Proof.** We prove Lemma 5.3 in two steps. We first show that each finite length segment $(p(), ... , p())$ can be generated by starting with the first fact $p()$ and repeatedly applying $T$ (Lemma 5.4). We then show that any segment $(p(), ... , p())$ produced by applying $T$ in the above manner preserves the well-founded property associated with $p$ (Lemma 5.5). This concludes the proof of Lemma 5.3.

**Lemma 5.4.** For each finite segment $(p(), ... , p())$ in a path of a derivation tree, the corresponding argument mapping can be generated by starting with the node $p()$ and applying transformation $T$ a finite number of times. Furthermore, a transformation tree can be constructed describing this generation with root $p( )$ for an arbitrary property $Prp$.

**Proof.** We prove the claim by induction on the length of the given segment.

**Induction Hypothesis.** The claim is true for all segments of length less than $k$.

**Basis.** $k = 2$: this must be a simple segment and for all segments that are simple cycles, the claim holds trivially. The construction of the transformation tree is simple and is essentially described in the induction step.

**Induction Step.** $k > 2$. For all segments that are simple cycles, the claim holds trivially. So, assume the segment is not simple. Consider a segment $(p(), ... , p())$ of length $k$ that is not a simple cycle. We prove the claim in two steps. First, we prove it for segments in which $p$ occurs only at the endpoints. Next, we show that it holds for arbitrary segments.

Suppose that the given segment $P$ only contains $p$ at the endpoints. Scanning from left to right, let $q_1$ be the first predicate that appears again later in the segment. Consider the subsequence from the leftmost occurrence of $q_1$ to the rightmost occurrence of $q_1$ in the given segment and denote this subsequence as $Q_1$. Let the next predicate (between the rightmost occurrence of $q_1$ and the right end of the sequence) repeat be $q_2$. Define the subsequence $Q_2$ as before, and so on for all repeating predicates. It is easy to see that the given segment $P = (p(), ... , p())$ can be represented as $(p(), N1, q1, N2, q2, ... , p())$. We claim that there must be a simple cycle $C$ of argument mappings of the form $(p(), N1, q1, N2, q2, ... , p())$. The claim is easy to verify.
by construction, no node in any segment $N_i$ appears in any segment $N_j$, $i > j$; none of the $q_k$ nodes appears in any $N_i$; and $q_i(q) > q_j$ if $i > j$. An (argument mapping) cycle of this form can be obtained from the given segment $(p( ), Q_1, Q_2, Q_3, ..., p( ))$ by simply merging the first and last occurrence of each $q_i(q)$ atom, for all $i$. These first and last occurrences also identify the rules to be used in the composite argument mapping. So, the composite argument mapping reflects precisely the derivation sequence in the derivation tree.

Now, since $Q_1$ denotes a derivation tree path segment $(q_1( ), ..., q_1( ))$ of length less than $k$, by the induction hypothesis, the corresponding argument mapping can be generated by starting with $q_1( )$ and repeatedly applying transformation $T$. Indeed, such a sequence of applications will never generate a node of the form $p( )$, since $Q_1$ does not contain any occurrences of $p$. We can similarly generate the segments $Q_i$, $i > 1$, in order, by expanding the corresponding facts $q_i(q)$.

A transformation tree describing the transformation can be built as follows. The root of the transformation tree $T_p$ that we construct is $p( )^{(1)}_{(1)}, Prp$, where $Prp$ is the property associated with $p$. The children of the root are the nodes $p( )^{(1)}, Prp, Q_1, q_1( )^{(1)}, Prq_1, Q_2, q_2( )^{(1)}, Prq_2, ..., p( )^{(1)}, Prp$, where $Prq_i$ is obtained using the procedure call $tag\_prop\_ Cic\_qi\_qk$. (For simplicity, we have not expanded the nodes corresponding to the segments $N_i$, but this is straightforward to do.) Inductively, for each $Q_i$ there is a transformation tree $T_i$ rooted at $q_i^{(i)}, Prq_i$ describing the generation of $Q_i$. Since $p( )$ does not occur in $Q_i$, the resulting transformation tree can be changed by adding $p( )$ to the first argument of each superscript. The result is still a legal transformation tree. So, we may replace in $T_p$ each child $q_i^{(i)}, Prq_i$ of $p( )^{(1)}, Prp$ by the tree $T_i$.

The result is a legal transformation tree of the whole generation process. Thus, we have shown how the given segment of length $k$ can be generated by repeated applications of $T$ that obey the restriction in the definition of transformation trees; indeed, we have shown how to construct the corresponding transformation tree $T_p$.

Finally, consider the case of segments $P$ in which $p$ appears at least once in the middle. We can represent the given segment $P$ as $(p( ), Q, p( ), ..., p( ))$, where $Q$ does not contain any occurrences of $p$. The argument mapping corresponding to the subsequence $(p( ), Q, p( ))$ can be generated from $p( )$ by applications of $T$ that obey the restrictions, from the induction hypothesis. The rest of the given segment can subsequently be generated from the second to the leftmost occurrence of $p( )$, again from the induction hypothesis. The construction of the transformation tree $T_p$ for this case is a straightforward combination of the transformation trees for the subsequences discussed above.

This concludes the proof of Lemma 5.4.

In the above proof, we not only showed that argument mappings corresponding to path segments in derivation trees can be constructed by applications of $T$, we also provided a construction of the corresponding transformation tree. We will refer to transformation trees generated using this construction as canonical transformation trees.

**Lemma 5.5.** Suppose that check\_clique $(C)$ returns true. If a segment $(p( ), ..., p( ))$ of a derivation tree can be generated from a node $p( )$ by applications of $T$ corresponding to a canonical transformation tree, then the segment preserves the well-founded property for $p$.

**Proof.** We prove the following (stronger) claim by induction on the number $k$ of applications of $T$: Suppose that Check\_clique $(C)$ returns true. Then, if a segment $(p( ), ..., p( ))$ of a derivation tree is produced from a fact $p( )$ by applications of $T$ corresponding to a canonical transformation tree, then the segment preserves the well-founded property (say $prop_p$) for $p$, and

2. Consider the canonical transformation tree for these applications. For each node $q_i^{(i)}, Prq_i$, there is a call $verify(q, Iq, Prq_i)$ in the execution of $verify(p, \{ \}, prop_p)$, and all these calls return true. (We note that if any call in $verify$ returns false, then false is returned overall.)

3. The above two claims imply that the canonical transformation tree of the transformation steps has no node whose second entry in the superscript is false. This is because a call to $verify(q, Iq, false)$ generates, for some $Ci$, a call to $check\_prop(Ci, false)$ and this call would return false, since the checked property is false and there is no way to satisfy a property that is false.

**Induction Hypothesis.** The claim holds for all segments $(p( ), ..., p( ))$ produced from a node by $k$ applications of $T$.

**Basis.** For $k = 1$, consider the simple cycle $C = (p( ), ..., p( ))$. This corresponds to the call $verify(p, \{ \}, prop_p)$. The call $check\_prop(C, prop_p)$ in Step 1 of $verify$ ensures that this cycle preserves the well-founded property associated with $p$. This proves part (1) of the claim for the basis case. There is a call $tag\_prop(C, prop_p, q, prop_p)$ in Step 2 of $verify$ that associates the tag $prop_p$, for each node $q$ in this cycle. Also, the superscript for node $q$ in the transformation tree is $(\{ p( ), prop_p \})$, by construction of the transformation tree. There is a call $verify(q, \{ \}, prop_p)$ in Step 2 of $verify(p, \{ \}, prop_p)$, proving part (2) of the claim for the basis case.

**Induction Step.** Every segment produced by $k + 1$ applications of $T$ must be produced by replacing some node in a segment produced by $k$ applications. Consider a segment $C = (p( ), ..., p( ))$ produced by $k$ applications of $T$. By the hypothesis, it preserves the well-founded property for $p$. Further, in the corresponding transformation tree, nodes
are tagged with ignore sets and properties such that the induction hypothesis holds. Let a node \( q \) (tagged with \((Iq, prop_q)\)) in the segment \( C \) be replaced by a simple cycle \( C1 = (q_1, \ldots, q_l) \). There is a call \( \text{verify}(q, Iq, prop_q) \) according to (part 2) of the induction claim in the hypothesis. This generates a call \( \text{check} \cdot \cdot \cdot \) for every simple cycle \( Cx \) of the form \((q_1, \ldots, q_l) \) that does not contain any node in \( Iq \). In particular, since \( C1 \), by the restriction of transformation \( T \), cannot contain nodes in \( Iq \) there is a call \( \text{check} \cdot \cdot \cdot \). Thus, the segment \( C1 \) preserves \((q, prop_q)\). From the specification of \( \text{tag} \cdot \cdot \cdot \), this ensures that the segment \( C \), after the replacement of \( q \) by \( C1 \), still preserves \((p, prop_p)\). This proves part (1) of the claim.

For each nodes \( q \) in \( C1 \), the call to \( \text{tag} \cdot \cdot \cdot \) in Step 2 of the call \( \text{verify}(q, Iq, prop_q) \) associates a property \( prop_{p,q} \), and the corresponding node in the transformation tree has the tag \((Iq \cup \{q\}, prop_{p,q})\), by construction of the transformation tree. Further, in Step 2 of the call \( \text{verify}(q, Iq, prop_q) \), the call \( \text{verify}(1, Iq \cup \{q\}, prop_{p,q}) \) is generated, proving part (2) of the claim.

This completes our proof of Lemma 5.5.

The following result summarize our approach.

**Corollary 5.6.** Consider a clique \( C \) in a database \( DB = (EDB, IDB, IC) \). If \( \text{check} \cdot \cdot \cdot \) returns true then every predicate in the clique is strongly safe.

**Proof.** This follows immediately from Theorems 5.1 and 5.2. (Recall that Theorem 5.1 guarantees strong safety given a constant that bounds the height of all derivation trees. Theorem 5.2 asserts the existence of such a constant if \( \text{check} \cdot \cdot \cdot \) returns true.)

5.1. Refinements of the Approach

We now consider some limitations of procedure \( \text{check} \cdot \cdot \cdot \) and the closely related Theorem 5.2 and show how it can be strengthened to overcome them.

**Example 5.2.** We have not considered any techniques for propagating monotonicity constraint or boundedness information. This can prevent us from inferring safety in some cases, as the following example illustrates:

1. \( p(X, Y) := q(X, Y), q(2(X, Y)) \).
2. \( q(1(X, Y)) := b(X, Y) \).
3. \( q(2(X, Y)) := f(U, X), p(U, Y) \).

\( b \) is a finite base predicate and \( f \) is an infinite base predicate. The first argument of \( p \) is bounded since it can only take values in \( b \). However, \( \text{check} \cdot \cdot \cdot \) considers whether this property is preserved by the rule cycle (3, 1) corresponding to the argument mapping \((p, q2, 3)\). Since this cycle considered in isolation does not preserve boundedness of the first argument of \( p \), by using Theorem 5.2 we cannot show that the first argument of \( p \) is bounded. We additionally require techniques that would allow us to infer that the first argument position of \( q1 \) is bounded and to include this information in the body of rule 1.

Similarly, examples can be constructed in which additional techniques for inferring monotonicity constraints are required to establish strong safety using \( \text{check} \cdot \cdot \cdot \). Nevertheless, we would like to emphasize the extensible nature of our approach.

1. As the previous example demonstrated, we need to identity bounded argument positions and to propagate this information. There are some obvious ways of doing this. For example, the atom \( X < 10 \) in a rule tells us that \( X \) is bounded above, and \( b(X) \), where \( b \) is a finite base relation, tells us that \( X \) can only range over a finite set of values. Recall that the methods presented in this paper focus on strong safety. Other methods for inferring safety can be used in our analysis to identify arguments that are bounded. By adding this information to the bodies of rules in the form of additional atoms we can extend the scope of \( \text{check} \cdot \cdot \cdot \). We can also rewrite rules to propagate equality information, as described in [13].

2. We have not addressed the issue of how to infer monotonicity constraints. Work in this direction has been presented in [14, 6, etc.], and we can take advantage of those results to infer constraints.

3. Two programs, \( \text{check} \cdot \cdot \cdot \) and \( \text{tag} \cdot \cdot \cdot \) are assumed to be given as inputs to \( \text{check} \cdot \cdot \cdot \). Using more sophisticated versions of these programs that can deal with a richer class of well-founded orderings clearly increases the scope of \( \text{check} \cdot \cdot \cdot \).

We now consider a more fundamental generalization of procedure \( \text{check} \cdot \cdot \cdot \) and Theorem 5.2.

**Example 5.3.** The following example is similar to Example 5.2:

1. \( p(X, Y) := q1(X, Y), q2(X, Y), X > 0 \).
2. \( q1(X, Y) := g(X, U), p(U, Y) \).
3. \( q2(X, Y) := f(U, X), p(U, Y) \).

\( g \) and \( f \) are infinite base predicates, and all FCs hold over their arguments. Further, \( g_1 < g_2 \) holds. By examining the argument mapping \((p(U, Y), q1(X, Y), 2), (q(X, Y), 1) \), we see that the first arguments of \( p \) is monotonically decreasing. Theorem 5.2 cannot show this because it considers the argument mapping \((p(U, Y), q2(X, Y), 3), (q2(X, Y), p(U, Y), 1) \) and the corresponding simple rule cycle (1, 3), which does not preserve this property. So Theorem 5.2 cannot “see” that in fact the potential problem if the (3, 1) rule cycle is in fact “handled” by the presence of the (2, 1) rule cycle and that this program is strongly safe.
The previous example brings out a limitation of Theorem 5.2 as a practical tool. The intuition behind this can be explained as follows. The theorem seeks to examine every path in a derivation tree and to show that it is bounded in length (by a constant dependent upon the database and clique rules). This is sufficient to ensure that no rule can be applied an unbounded number of times. This in turn ensures that the predicate labeling the root is safe. However, it may not be possible to show such a bound by examining a path independently of the rest of the derivation tree in which it appears. The following simple theorem indicates how we can strengthen the approach by considering variants of the given program. We discuss how check_clique can be modified to do this efficiently following the presentation of the theorem.

DEFINITION. Let us define the head of a rule to be covered by a set of body atoms if Assumptions (1) and (2) about programs are still satisfied when we delete the remaining body atoms.

DEFINITION. Consider a clique $C'$ that is obtained from a clique $C$ by modifying each rule as follows: Choose a set of covering body atoms and delete the remaining body atoms. We call $C'$ a variant of $C$.

THEOREM 5.6. If check_clique($C'$) returns true for some variant $C'$ of $C$, then every predicate in clique $C$ is strongly safe.

Proof. Assumptions (1) and (2) about programs are satisfied by $C'$. By the correctness of check_clique, $C'$ is strongly safe if check_clique($C'$) returns true.

Every predicate in $C$ also appears in $C'$. Further every rule of $C$ is a rule of $C'$, possibly with some additional body atoms. The additional atoms can only further restrict the set of computed tuples further. Since every predicate in $C'$ is safe, it follows that every predicate in $C$ is also safe. That is, $C$ is strongly safe.

Now let us consider how to refine check_clique using the above theorem. Consider a clique $C$ containing a predicate $p$, and let $r$ be a rule defining $p$: $p(\ldots) :- q_1(\ldots), q_2(\ldots), \ldots, q_n(\ldots)$.

Suppose that there is a variant $C'$ of this clique such that check_clique($C'$) returns true. Consider the covering subset of body atoms, say $Q$, in rule $r$. By construction, all other body atoms in $r$ are deleted in order to obtain the corresponding rule, say $r'$ in $C'$. Let prop be the well-founded property associated with $p$ in check_clique($C'$). For every atom $q_i(\ldots)$ in $Q$, every simple cyclic composite argument mapping $(q_1(\ldots), p_1(\ldots) \ldots, (p_l(\ldots), \ldots))$ must satisfy property prop; else check_clique($C'$) would not have returned true. Let us call this set of mappings the set of covering mappings for $r$ and call all other simple cyclic composite argument mappings covered mappings.

We can make the following modification to the verify procedure. As we test simple composite argument mappings of the form $(p_1, \ldots, q_2(), \ldots, (q_j, p, r))$, if we find a subset of these mappings that is a covering set of mappings, we can ignore covered mappings having the same form. (This is similar to the way we ignore mappings that contain a node in the set IGNORE.) On the other hand, we may encounter one of the other simple cyclic composite argument mappings (of the form $(p_1, \ldots, q), (q_j, p, r)$, for which the function check_prop fails) before we have tested all mappings in some covering subset of mappings (assuming that such a subset exists). So if verify fails for a simple cyclic composite argument mapping, rather than aborting, this should be recorded and verify should continue testing other mappings till all candidate $p_i$’s (from the corresponding rule $r$) have been checked. If a “good” subset is found, verify succeeds, and otherwise it fails. The details of the modification are left to the reader. (Recording the failure of verify allows us to avoid repeating the test if it is required again subsequently.)

We remark that the refined version of check_clique that uses Theorem 5.6 is not only more powerful than the previous version, it is often considerably more efficient. For example, consider the case when the body atoms covering the head are non-recursive. The safety test then becomes trivial, since, given our three assumptions about programs; non-recursive programs are always safe and effectively computable.

In concluding this sections, we observe a weakness of the framework that points to a direction for future work. The clique by clique approach, which is necessary to deal efficiently with large programs, may fail to detect (strong) safety of some (strongly) safe programs. Intuitively, this happens when a (sub)goal is (strongly) safe, but the corresponding predicate is not. (This is also a problem with other approaches, such as capture rules, for instance.) For example, consider the query $p(X, Y) :- f(X), b(Y)$. $f$ is an infinite base predicate and $b$ is a finite base predicate. The predicate $p$ is unsafe since the rule generates an infinite number of tuples for $p$. However, the query is safe since the rule can generate at most $N$ tuples with the same value in each argument, where $N$ is the number of tuples in the finite relation $b$. The reason we cannot deal with such an example has to do partly with the rewriting strategies. These strategies are used to propagate bindings in the query into the rules defining the query, but, as presented in [5], they cannot propagate equalities between free variables, for example. The given query can be shown safe using the following extension to our framework. In considering $p$, although it is not safe, we can see that the second argument
is bounded. This information can be propagated into the clique in which the query \( p(X, X)_1 \) appeared by adding an atom \( \text{finite}(X) \) to the rule containing the atom \( p(X, X)_1 \). Unfortunately, this is not a complete solution, as the following change to the definition of \( p \) demonstrates

\[
p(X, Y) := f(X), b(Y).
p(X, Y) := f(Y), b(X).
\]

The query \( p(X, X)_1 \) is still safe, but we cannot show this, even with the extension suggested above. (Since neither argument of \( p \) is a priori bounded, we cannot add an atom of the form \( \text{finite}(...) \)—the reason that the query is finite is that there are only a finite number of \( p \) facts with the same value in both argument positions, even though projecting on either argument position would yield an infinite number of values.)

6. SIPS, ADORNED PROGRAMS, AND MAGIC PROGRAMS

In this section, we briefly review the notions of sideways information passing, adornments and the Generalized Magic Set program transformation. The reader is referred to \cite{sideways} for additional details.

6.1 Sideways Information Passing

A sideways information passing strategy, henceforth referred to as a sip, is an inherent component of any query evaluation strategy. A sip characterizes a top-down propagation of bindings. Informally, for a rule of a program, a sip describes how bindings passed to the head by unification are used to evaluate the predicates in the body. It represents a decision about the order in which the predicates of the rule are evaluated and how values for variables are passed from predicates to other predicates during evaluation. Consider, for example, the ancestor query presented in Section 1. The first argument is bound to \( \text{john} \), and by unification, the variable \( X \) in the second rule is bound to \( \text{john} \). We can evaluate \( \text{par} \) using \( \text{par} \) and obtain a set of bindings for \( Z \). These are passed to \( \text{anc} \) to generate subgoals, which in this case have the same binding pattern. This is in fact the way in which a top-down strategy like that employed by the programming language Prolog would compute this query.

Generalizing from this example, we may say that the basic step of sideways information passing is to evaluate a set of predicates (possibly with some arguments bound to constants) and to use the results to bind variables appearing in another predicate. The order in which predicates are solved and the bindings are passed is determined as a consequence of the control strategy in top-down methods. We try to separate this order from the flow of control, leading to the definition of a sip as a labeled graph, below.

Let \( r \) be a rule, with head atom \( p(\theta) \), and let \( p\ldots h \) be a special predicate, denoting the head predicate restricted its occurrence, starting from 0. (The numbering is meant to identify the positions in the rule. It is irrelevant when unification with heads of other rules is considered.) Let \( P(r) \) denote the set of predicate occurrences in the body. A sip for \( r \) is a labeled graph that satisfies the following conditions:

1. Each node is either a subset or a member of \( P(r) \cup \{ p\ldots h \} \).
2. Each arc is of the form \( N \to q \), with label \( \gamma \), where \( N \) is a subset of \( P(r) \cup \{ p\ldots h \} \), \( q \) is a member of \( P(r) \), and \( \gamma \) is a set of variables, such that each variable of \( \gamma \) appears in some member of \( N \).

These two conditions define the nature of nodes and arcs of a sip. The following condition provides a consistency restriction on a sip. For a graph with nodes and arcs as above, define a precedence relation on the members of \( P(r) \cup \{ p\ldots h \} \) as

1. \( p\ldots h \) precedes all members of \( P(r) \).
2. A predicate that does not appear in the graph, follows every predicate that appears in it.
3. If \( N \to q \) is an arc and \( q' \in N \), then \( q' \) precedes \( q \).

We can now state the last condition defining a sip:

3. The precedence relation defined by the sip is acyclic.

Example 6.1. Consider the following rules, known as the same-generation program:

\[
\begin{align*}
\text{sg}(X, Y) &:= \text{flat}(X, Y). \\
\text{sg}(X, Y) &:= \text{up}(X, Z1), \text{sg.1}(Z1, Z2), \text{flat}(Z2, Z3), \text{sg.2}(Z3, Z4), \text{down}(Z4, Y). \\
\text{Query: } &\text{sg}(\text{john}, Z)_1?
\end{align*}
\]

We have numbered the \( \text{sg} \) occurrences in the second rule for convenience. The \( \text{sg} \) predicate is the only derived predicate, and all others are base predicates.

Given the query, the natural way to use the second rule seems to be to solve the predicates in the indicated order, using bindings from each predicate to solve the next predicate. This information passing strategy may be represented by the following sip:

\[
\begin{align*}
\{ \text{sg} \ldots h \} &\to _x \text{up}; \\
\{ \text{sg} \ldots h, \text{up} \} &\to _{Z1} \text{sg} \ldots 1; \\
\{ \text{sg} \ldots h, \text{up}, \text{sg} \ldots 1 \} &\to _{Z2} \text{flat}; \\
\{ \text{sg} \ldots h, \text{up}, \text{sg} \ldots 1, \text{flat} \} &\to _{Z3} \text{sg} \ldots 2
\end{align*}
\]

This is a simplification of the definition in \cite{sideways}.
We have only indicated how bindings are propagated into derived predicates; we will not concern ourselves with bindings for base predicates, although these are useful for selecting appropriate access methods.

6.2. The Adorned Rule Set

Intuitively, an adorned occurrence of the predicate, \( p' \), corresponds to a computation of the predicate with some arguments bound to constants, and the other arguments are free, where the bound arguments are those that are so indicated by the adornment \( a \). For example, \( p^{\text{so}} \) corresponds to computing \( p \) with the first two arguments bound and the last argument free.

Let a program \( P \) and a query \( q(e, X) \) be given, where \( e \) is the vector of bound arguments and \( X \) is the vector of free arguments. \( q \) is called the query predicate. We construct a new, adorned version of the program, denoted by \( P^{\text{ad}} \). In the construction we replace derived predicates of the program by adorned versions, where for some predicates we may obtain several adorned versions. For each adorned predicate \( p^{\text{ad}} \) and for each rule with \( p \) as its head, we choose a sip and use it to generate an adorned version of the rule (the details are presented below). Since the head of a rule may appear with several adornments, it follows that we may attach several distinct sips to versions of the same rule, one to each version.

We maintain a set of “unmarked” adorned predicates and use this set to guide our generation of adorned rules. Initially, this set contains \( q^{e} \), which is the query predicate \( q \) with adornment \( e \), in which precisely the positions bound in the query are designated as \( b \); and the adorned predicate \( p^{\text{ad}} \), for each predicate \( p \) in the program. In general, if \( p^{\text{ad}} \) is an adorned predicate, then for each rule that has \( p \) in its head, we generate an adorned version for the rule and add it to \( P^{\text{ad}} \). The adorned version of a rule may generate additional adorned predicates. We then mark \( p^{\text{ad}} \). The process terminates when no unmarked adorned predicates are left. Termination is guaranteed, since the number of adorned versions of predicates for any given program is finite.

Let \( r \) be a rule in \( P \) with head predicate \( p \). We generate an adorned version, corresponding to an adorned predicate \( p^{\text{ad}} \), as follows: The new rules has \( p^{\text{ad}} \) as its head. Choose a sip \( s \), for the rule that matches the binding \( a \), i.e., the special predicate \( p_{-h}.h \) is the head restricted to arguments that are designated as bound in the adornment \( a \). Next, we replace each derived predicate in the body of the rule by an adorned version (and if this version is new, we add it to our collection of unmarked adorned predicates). We obtain the adorned version of a derived predicate in the body of the rule as follows. For each occurrence \( p_{i} \) of such a predicate in the rule let \( X_{i} \) be the union of the labels of all arcs coming into \( p_{i} \). (If there is no arc coming into \( p_{i} \), let \( X_{i} \) denote the empty label.) We replace \( p_{i} \) by the adorned occurrence \( p_{i}^{\text{ad}} \), where an argument position of \( p_{i} \) is bound in \( a_{i} \) only if it contains a variable that also appears in \( X_{i} \). (Recall that in Extended Datalog, terms are variables.)

Example 6.2. The following is the adorned rule set corresponding to the non-linear same generation example, for sip (1).

1. \( sg(X, Y) :- flat(X, Y). \)
2. \( sg(X, Y) :- up(X, Z), sg(Z, Z2), flat(Z2, Z3),
   sg(Z3, Z4), down(Z4, Y). \)
   Query: \( sg(john, Y)? \)

We will use these adorned rules to illustrate the rule rewriting algorithms presented later.

6.3. Generalized Magic Sets

The next stage in the proposed transformation is to define additional predicates that compute the values that are passed from one predicate to another in the original rules, according to the sip strategy chosen for each rule. These auxiliary predicates are called magic predicates and the sets of values that they compute are called magic sets. Intuitively, these values correspond to goals in a top-down execution. Each of the original rules is modified so that only facts that are answers to one of these “goals” are generated. The intention is that the bottom-up evaluation of the modified set of rules simulate the sip we have chosen for each adorned rule, thus restricting the search space.

The transformation consists of the following:

1. We create a new predicate \( \text{magic}(. . p) \) for each \( p^{\text{ad}} \). The arity of the new predicate is the number of occurrences of \( b \) in the adornment \( a \), and its arguments correspond to the bound arguments of \( p^{\text{ad}} \).
2. For each rule \( r \) in \( P^{\text{ad}} \), and for each occurrence of an adorned predicate \( p^{\text{ad}} \) in its body, we generate a magic rule defining \( \text{magic}(. . p) \) (see below).
3. Each rule is modified by the addition of a magic atom to its body.
4. We create a seed for the magic predicates, in the form of a fact, obtained from the query. The seed provides an initial value for the magic predicates. Using our notation above, the seed is \( \text{magic}(. . q^{e}(c)) \), where \( c \) denotes the vector of constants for the bound arguments in the query.

(We follow the convention that the seed, which is generated from a specific query, is not part of the rewritten program, but is added to it before execution.)
We now explain the second step in more detail. We use the following notation. Greek letters (possibly numbered using subscripts) are used to denote argument lists. If \( \gamma \) denotes the argument list of a predicate \( p^a \), then \( \gamma' \) (resp. \( \gamma'' \)) denotes \( \gamma \) with all arguments that are bound (resp. free) in adornment \( a \) deleted. Consider the adorned rule:

\[
r: p^a(\gamma) : q^a_1(\theta_1), q^a_2(\theta_2), \ldots, q^a_n(\theta_n)
\]

Let \( s \) be the sip associated with this rule. Assume that the predicates in the body are ordered, according to the sip, i.e., those that participate in the sip precede those that do not, and the predicates in the tail of an arc precede the predicate at the head of the arc.

Consider \( q_i \). If \( N \rightarrow q_i \) is the only arc entering \( q_i \) in the sip, we generate a magic rule defining \( \text{magic} \cdot q_i^a \) as follows. The head of the magic rule is \( \text{magic} \cdot q_i^a(\theta_i^0) \). If \( q_j, j < i, \) is in \( N \), we add \( q_j^a(\theta_j) \) to the body of the magic rule. If the special predicate denoting the bound arguments of the head is in \( N \), we add \( \text{magic} \cdot p^a(\chi') \) to the body of the magic rule.

If there are several arcs entering \( q_i \), we define the magic rule defining \( \text{magic} \cdot q_i^a \) in two steps. First, for each arc \( N_j \rightarrow q_i \), with label \( \chi_j \), we define a rule with head \( \text{label}_{-j} \cdot q_i \cdot \chi_j \). The body of the rule is the same as the body of the magic rule in the case where there is a single arc entering \( q_i \) (described above). Then the magic rule is defined as follows: The head is \( \text{magic} \cdot q_i^a(\theta_i^0) \). The body contains \( \text{label}_{-j} \cdot q_i \cdot \chi_j \) for all \( j \) (i.e., for all arcs entering \( q_i \)).

In the third step, we modify the original rule by inserting an atom \( \text{magic} \cdot p^a(\chi') \) into the body of the rule \( r \) as a “guard.” Intuitively, this “magic” relation contains the values that may passed to this rule when invoked with a \( p^a \) goal. In a bottom-up evaluation, the rule does not succeed unless the appropriate values are first computed and, thus, the computation is restricted to the solution of relevant goals.

**Example 6.3.** Using the sips presented earlier, the Generalized Magic Sets strategy rewrites the adorned rule set corresponding to the non-linear same generation example into the following set of rules. (The rule numbers refer to the adorned rule set.)

\[
\begin{align*}
\text{magic} \cdot s_{g^b}(\text{john}). & \quad \text{[Seed; from the query rule]} \\
\text{magic} \cdot s_{g^b}(X_1) & \mapsto \text{magic} \cdot s_{g^b}(X), \text{up}(X, Z_1). & \quad \text{[From rule 2, 2nd-body atom]} \\
\text{magic} \cdot s_{g^b}(Z_3) & \mapsto \text{magic} \cdot s_{g^b}(X), \text{up}(X, Z_1), s_{g^b}(Z_1, Z_2), s_{g^b}(Z_2, Z_3). & \quad \text{[From rule 2, 4th-body atom]} \\
\end{align*}
\]

Let \( P_{\text{me}} \) denote a program obtained from \( P_{\text{md}} \) by the transformation above. We now consider the correctness of the transformation. Recall that an adorned predicate \( p^a \) represents queries of the form \( p(i) \) in which all arguments corresponding to \( b^i \)’s in adornment \( a \) are assigned constants.

For two programs \( P_1 \) and \( P_2 \) and a query form \( p^a \), we say that \( (P_1, p^a) \) and \( (P_2, p^a) \) are equivalent if for any assignment of constants to the arguments of \( p^a \) that are bound in \( a \), the two programs produce the same answer for the resulting queries.

**Theorem 6.1** [\( \ast \)]. For each \( p^a \) that appears in \( P_{\text{md}} \), \( (P_1, p^a) \) is equivalent to \( (P_{\text{me}}, p^a) \).

**7. Analyzing Programs Obtained from the Magic Sets Transformation**

Given a program \( P \), we use a set of sips, one sip per rule per adornment of the head predicate (which we assume are also given) to produce an adorned program \( P_{\text{md}} \) and then the program \( P_{\text{me}} \). We will henceforth assume the following:

1. Consider a rule in \( P_{\text{md}} \). Let \( s \) be the sip associated with it. The rule is written so that for each arc \( N \rightarrow p \) in \( s \), the adorned version of \( p \) in \( P \) appears to the left of \( C \), and \( p^a \) in \( P_{\text{md}} \), each appearing to the left of \( C \) in \( \alpha \), guarantees that this is possible.

2. For every adorned occurrence \( f^a \) of an infinite base relation \( f \), the set of bound arguments determines all other arguments (using the FCs given to hold for \( f \)).

Thus, the constants for a bound argument in a predicate can be determined from the predicates to the left of the given predicate. These two assumptions therefore guarantee assumption (2) about programs (from Section 3). Assumptions (1) and (3)—which concern head variables and effective computability for infinite relations—about programs are also assumed to hold.

We now introduce some definitions. Recall that a rule is said to be strongly safe if every predicate that appears in the body is strongly safe. Consider a clique \( C \) in \( P_{\text{md}} \). For each adorned predicate \( p^a \) in \( C \), consider the rules in \( P_{\text{md}} \) with \( p^a \) as the head predicate. We say that these rules in \( P_{\text{md}} \) are generated from clique \( C \). Also, recall that for each adorned predicate \( p^a \) in \( C \), these magic rules are also said to be generated from clique \( C \).
We test for strong safety as follows. Given a program \( P_{\text{mg}} \), we test it according to the ordering of cliques in \( P_{\text{mod}} \), beginning with cliques that are leaves. Note the difference in the way we treat such programs—in general, we would proceed according to the ordering of cliques in the given program. This special treatment of "magic" programs is necessary because the Magic Sets transformation introduces rules that often combine cliques in the original program into larger cliques in the rewritten program.

**Example.** Consider the following program:

\[
p(X, Y) : \neg p(X, Z), q(Z, Y).
q(X, Y) : \neg q(X, Z), d(Z, Y).
q(X, Y) : b(X, Y).
\]

It has two cliques with recursive predicates, containing \( p \) and \( q \), respectively. Given a query \( q(5, Y) \ldots \) and a left to right order for all sips, the adorned program is identical to the original program except that all occurrences of \( p \) and \( q \) are replaced by \( p^{bf} \) and \( q^{bf} \). The Magic Sets transformation yields the following program:

\[
p^{bf}(X, Y) : \text{magic} \cdot p^{bf}(X), p^{bf}(X, Z), q^{bf}(Z, Y).
q^{bf}(X, Y) : \text{magic} \cdot q^{bf}(X), q^{bf}(X, Z), d(Z, Y).
q^{bf}(X, Y) : \text{magic} \cdot q^{bf}(X), b(X, Y).
\text{magic} \cdot q^{bf}(Z) : \text{magic} \cdot p^{bf}(X), p^{bf}(X, Z).
\text{magic} \cdot p^{bf}(5).
\]

The rule for \( \text{magic} \cdot q^{bf} \) contains \( p^{bf} \) in the body, and the rules for \( q^{bf} \) contain \( \text{magic} \cdot q^{bf} \) in the body. Thus, the two cliques in the adorned program are combined into a single clique in the rewritten program.

So, the sizes of cliques in the rewritten program are likely to be large, which implies that the cost of testing for safety on the basis of cliques in \( P_{\text{mg}} \) is likely to be very high (since the cost is exponential in the size of the clique). In this section, we present results that enable us to test \( P_{\text{mg}} \) according to the structure of cliques in \( P_{\text{mod}} \). (In general, the adorned program can contain several adorned versions of each derived predicate, and thus, of each rule. However, the clique structure in \( P_{\text{mg}} \) is still similar to that in the original program, and we therefore avoid the problem of large cliques in \( P_{\text{mg}} \).)

We now present a brief overview of the results in this section. Intuitively, our approach is to proceed on a clique by clique basis, at each step reducing the size of the clique structure by eliminating leaf level cliques. Theorem 7.1 allows us to deal with leaf cliques in \( P_{\text{mod}} \). Theorem 7.3 allows us to deal with non-leaf cliques in \( P_{\text{mod}} \) by showing how predicates from other cliques can be replaced by base predicates in the safety analysis. Theorem 7.4 is also intended to deal with non-leaf cliques and may sometimes be applicable when Theorem 7.3 is not. The other lemmas are either technical lemmas used in proofs or weaker forms of these theorems developed for expository purposes.

Consider \( P_{\text{mg}} \). We now identify two subsets of rules that are referred to extensively in the rest of this section. Let \( C \) be a clique in \( P_{\text{mod}} \) and let \( q\alpha \), the query predicate, be in \( C \). (Note that the query predicate guides the generation of adorned and magic programs from a given program \( P \) although our notation—e.g., \( f^{p_{\text{mod}}} \)—does not make this explicit.) The set of rules in \( P_{\text{mod}} \) that are in \( C \) or some clique that is descendant of \( C \) is denoted as \( P_{\text{mod}, \text{mod}, C} \). Similarly, consider only the rules in \( P_{\text{mg}} \) that are generated from some rule in \( P_{\text{mod}, \text{mod}, C} \). We denote this subset of rules as \( P_{\text{mg}, \text{mod}, C} \).

As per our convention regarding seeds, we assume that a "seed" magic fact is added to the program before execution. When we say that a (magic or non-magic) predicate is safe in \( P_{\text{mod}, \text{mod}, C} \), we mean that is safe no matter what the choice of the seed is.

We will make the following assumption, without loss of generality, in the sequel: Suppose that \( p^* \) (for some adornment \( \alpha \) in \( P_{\text{mod}} \) resp. in clique \( C \), or a descendant of clique \( C \) in \( P_{\text{mod}} \)). If \( p_{\text{mg}} \) (resp. \( P_{\text{mg}, \text{mod}, C} \)) does not contain \( p^* \) for some predicate \( p \), as per the Magic Sets transformation, we will add rules to \( P_{\text{mg}} \) (resp. \( P_{\text{mg}, \text{mod}, C} \), defining \( p^* \) as follows: For each rule in the original program that defines \( p \), we generate a rule defining \( p^* \) in which the adornment of every body atom is also \( \alpha \).

The above assumptions are made primarily to simplify the presentation.

We have the following theorem describing conditions under which the predicate \( q \) in a leaf clique \( C_{\text{L}} \) can be replaced by a finite or infinite base predicate for each occurrence in another clique \( C_{\text{R}} \).

**THEOREM 7.1.** Consider the (topologically sorted) set of cliques over \( P_{\text{mod}} \). Let \( C_{\text{L}} \) be a leaf clique.

**Given:**

1. For every adorned predicate \( q_{\text{ai}} \) in clique \( C_{\text{L}} \), predicate \( q_{\text{ai}}^\alpha \) is safe in \( P_{\text{mg}} \).
2. For every adorned predicate \( q_{\text{ai}} \) in \( C_{\text{L}} \), magic\( \cdot q_{\text{ai}}^\alpha \) is safe in \( P_{\text{mg}, \text{mod}, C_{\text{L}}} \) if we replace all occurrences of adorned predicates (i.e., non-magic predicates) in \( C_{\text{L}} \) in the bodies of rules of \( P_{\text{mg}, \text{mod}, C_{\text{L}}} \) by occurrences of a finite relation.

**Then:**

All predicates in \( C_{\text{L}} \) and their magic predicates are strongly safe in \( P_{\text{mg}, \text{mod}, C_{\text{L}}} \).

**Proof.** The definition of \( q_{\text{ai}}^\alpha \) is the same in \( P_{\text{mg}} \) and \( P_{\text{mg}, \text{mod}, C_{\text{L}}} \). Further, \( q_{\text{ai}}^\alpha \) is always a subset of \( q_{\text{ai}}^\beta \) (since every rule for \( q_{\text{ai}}^\alpha \) is obtained by adding a body atom to a rule that defines \( q_{\text{ai}}^\beta \) and possibly changing the
adornments of some body atoms from \( ff \ldots f \) to “more restrictive” adornments.) By condition (1) of the hypothesis, it follows that each adorned predicate \( qi^{\text{ad}} \) in \( C_1 \) is safe in \( P_{\text{mod}} \) and in \( P_{\text{mod}.C_1} \).

Next, we turn to magic predicates in \( P_{\text{mod}.C_1} \). If each occurrence of a predicate \( qi^{\text{ad}} \) that is in \( C_1 \) is replaced by \( qi^{\text{ff}} \), since any relation \( qi^{\text{ad}} \) is a subset of the relation \( qi^{\text{ff}} \) (by the definition of adorned predicates), we clearly compute a superset of the facts that the original rules compute. And by condition (1), the replacing relation is finite. Thus, an immediate consequence of condition (2) of the hypothesis is that \( magic_{qi} \) is safe in \( P_{\text{mod}.C_1} \) for each predicate \( qi \) that is in \( C_1 \). Note that this argument relies only on the safety of \( q^{ff} \) and not its effective computability.

Now consider a predicate \( qi^{\text{ad}} \) in \( C_1 \). Since \( C_1 \) is a leaf clique, the only predicates that can define \( qi^{\text{ad}} \) in \( P_{\text{mod}.C_1} \) are predicates in \( C_1 \) and their magic predicates. (See comments following the proof.)

We have shown that all predicates in \( C_1 \) and their magic predicates are safe in \( P_{\text{mod}.C_1} \). Therefore, it follows that all predicates in \( C_1 \) and their magic predicates are strongly safe in \( P_{\text{mod}.C_1} \).

Consider the claim in the proof about the predicates that can define \( qi^{\text{ad}} \) in \( P_{\text{mod}.C_1} \). Note that such a claim would not hold for \( P_{\text{mod}} \). In general, \( qi^{\text{ad}} \) can be defined by magic predicates of predicates in \( C_1 \), and in \( P_{\text{mod}} \), there might be rules defining these magic predicates that contain predicates from other cliques. Thus, \( qi^{\text{ad}} \) could be defined by a predicate in another clique, through magic predicates of predicates in \( C_1 \). However, this cannot happen in \( P_{\text{mod}.C_1} \) since every magic rule in \( P_{\text{mod}.C_1} \) that defines \( magic_{qi} \), where \( qi \) is in \( C_1 \), is generated from \( C_1 \). Therefore, the body can only contain predicates from \( C_1 \) and possibly their magic predicates.

We observe that \( P_{\text{mod}.C} \) correctly computes all answers to a query of the form \( qi^{\text{ad}}(\vec{e}, \vec{X}) \). It follows that \( qi^{\text{ad}} \) in \( C_1 \), is replaced by \( \text{seed} fact magic_{qi}(\vec{e}) \). We make this claim precise in the following technical lemma, used in subsequent proofs.

**Lemma 7.2.** If \( q^a \) is in clique \( C \) then \( P_{\text{mod}.C} \cup \{ \text{seed for } q^a \} \) is a program that computes the same set of answers to a query \( q^a(\vec{e}, \vec{X}) \) (where \( \vec{e} \) and \( \vec{X} \) are vectors of constants and variables, respectively, corresponding to the bound and free arguments in adornments a) as the program \( P_{\text{mod}} \cup \{ \text{seed for } q^a \} \).

**Proof.** We note that \( P_{\text{mod}.C} \) can also be obtained by applying the Magic Sets transformation to \( P_{\text{mod}.C} \) with \( q^a \) as the query predicate, by definition of these two sets of rules. Also, \( P_{\text{mod}.C} \) contains all the relevant rules of \( P_{\text{mod}} \) to compute \( q^a \). By the correctness of the Magic Sets transformation, \( P_{\text{mod}.C} \cup \{ \text{seed for } q^a \} \) must therefore correctly compute all answers to a query of the form \( q^a(\vec{e}, \vec{X}) \) in \( P_{\text{mod}.C} \).

On the other hand, from the equivalence of \( P_{\text{mod}} \) and \( P_{\text{mod}} \cup \{ \text{seed for } q^a \} \) also correctly computes all answers to a query of the form \( q^a(\vec{e}, \vec{X}) \). It follows that \( P_{\text{mod}.C} \) and \( P_{\text{mod}} \) compute the same set of answers.

We now consider how to analyze cliques that are not leaves. We present a theorem that allows us (for the purpose of safety analysis) to replace predicates that do not belong to the clique by finite base relations, thus reducing the size of the clique structure. The idea, intuitively, is that we can think of a predicate \( q^a \) as a finite base predicate if the set of all \( q^a \) facts is finite and we have an effectively computable procedure of finding all answers to a \( q^a \) query. By virtue of this theorem, we fine some having to reason about the set of \( q^a \) queries; otherwise, this would require reasoning about the interaction of rules defining magic predicates with the rest of the program. In essence, we exclude from consideration precisely the magic rules that combine two cliques in \( P_{\text{mod}} \) into a single clique in \( P_{\text{mod}.C} \).

**Theorem 7.3.** Let the topologically sorted set of cliques over \( P_{\text{mod}} \) be represented as a directed acyclic graph (dag). Let \( C \) be a clique all of whose children are leaves.

**Given:**

1. For all \( Ci \) such that \( Ci \) is a child of \( C \), for all \( q^a \) in \( Ci \),
   (i) \( q^{ff} \) is safe in \( P_{\text{mod}} \), and
   (ii) \( q^a \) is strongly safe in \( P_{\text{mod}.Ci} \).
2. For all \( Ci \) such that \( Ci \) is a child of \( C \), for all \( q^a \) in \( Ci \), replace every occurrence of \( q^a \) in the body of a rule of \( P_{\text{mod}.C} \) that is not also in \( P_{\text{mod}.C} \) by a finite base predicate. Further, delete all rules defining \( magic_{q^a} \) that are obtained from such occurrences of \( q^a \). The program \( P \) obtained from \( P_{\text{mod}.C} \) by making these changes is strongly safe.

**Then:**

\( P_{\text{mod}.C} \) is strongly safe.

**Proof.** The proof consists of two parts. First, we prove that all predicates in \( C \) and their magic predicates are safe in \( P_{\text{mod}.C} \). To establish strong safety, we must also prove the safety in \( P_{\text{mod}.C} \) of predicates (and their magic predicates) in all children of \( C \). The safety of predicates in children cliques follows readily from condition (1) (and indeed is established in the course of part (1) of the proof). The second part of the proof consists of showing that the magic predicates corresponding to predicates in children cliques are also safe.

**Part (1)** From condition (1), the union of the answers to all subqueries on a predicate \( q^a \) in a clique \( Ci \) that is a
child of C, independent of the number of subqueries, is finite, since this union must be a subset of \( q^{\beta} \). Also, the answers to each subquery can be effectively computed: We may treat \( P_{\text{mg-mod.} C} \) as a program that is used to compute the answer to a subquery \( q^{\gamma}(c, \vec{X}) \), where \( q^{\gamma} \) is in \( C \). Condition (1)(ii) and Lemma 7.2 ensure that this subprogram will compute all the answers, for a given vector of values \( c \) (the seed). Termination and effective computability follow from Lemma 3.1.

Thus, if a predicate \( q^{\gamma} \) appears in a rule that is not in \( P_{\text{mg-mod.} C} \), we can replace it by a finite base relation for the purposes of the strong safety analysis. If we replace all such occurrences of \( q^{\gamma} \) by finite base relations, the resulting program \( P' \) is identical to program \( P \) of condition (2), except that \( P' \) contains some additional rules defining magic predicates \( magic \_ q^{\gamma} \). This difference is immaterial with respect to predicates that are not defined in \( P_{\text{mg-mod.} C} \), since they no longer depend on \( q^{\gamma} \), and therefore no longer depend on \( magic \_ q^{\gamma} \) either. In particular, if \( p' \) is in \( C \), neither \( p' \) nor \( magic \_ p' \) depends on \( q^{\gamma} \) or \( magic \_ q^{\gamma} \) in \( P' \) or \( P \). The safety in \( P' \), and therefore also in \( P_{\text{mg-mod.} C} \), of all predicates in \( C \) and their magic predicates now follows from condition (2).

Part (2) Let \( C_i \) be a clique in \( P_{\text{mg-mod.} C} \) that is a child of \( C \), and let \( q^{\gamma} \) be any predicate in \( C_i \). We must show that \( q^{\gamma} \) and \( magic \_ q^{\gamma} \) are strongly safe in \( P_{\text{mg-mod.} C} \). If we compare \( P_{\text{mg-mod.} C} \) and \( P_{\text{mg-mod.} C_i} \), the sets of rules defining \( q^{\gamma} \) are identical, and every rule that defines \( magic \_ q^{\gamma} \) in \( P_{\text{mg-mod.} C_i} \) is also present in \( P_{\text{mg-mod.} C} \). \( P_{\text{mg-mod.} C} \) contains some additional rules defining \( magic \_ q^{\gamma} \); this is exactly the set of rules deleted from \( P_{\text{mg-mod.} C} \) in obtaining \( P' \) (see condition (2)). We claim (see next paragraph) that the set of \( magic \_ q^{\gamma} \) tuples that can be generated by these additional rules in \( P_{\text{mg-mod.} C} \) is finite. From condition (2), it follows that \( magic \_ q^{\gamma} \) must be safe in \( P_{\text{mg-mod.} C} \).

It only remains to prove the claim made above. Consider the magic rules in \( P_{\text{mg-mod.} C} \) defining \( magic \_ q^{\gamma} \) (where \( q^{\gamma} \) is in some child \( C_i \) of clique \( C \) that were deleted in obtaining \( P' \). We prove that these rules generate only a finite number of distinct \( magic \_ q^{\gamma} \) tuples. Consider the correspondence between magic facts and subqueries. In essence, we must show that only a finite number of distinct subqueries is generated.

Consider a rule defining \( magic \_ q^{\gamma} \) (where \( q^{\gamma} \) is in \( C_i \) in \( P_{\text{mg-mod.} C} \)) that is generated from an occurrence of \( q^{\gamma} \) in \( C \). If there is an occurrence of a magic predicate \( magic \_ p^{\gamma} \) in the body of this rule, then \( p^{\gamma} \) must be in \( C \). (This property of the Magic Sets transformation and is easily verified.) We have shown that such magic predicates are safe in art (1) of the proof. Further, all non-magic predicates in the body of the rule are in \( C \) or in some \( C_j \) (which is a child of \( C \)). We have shown that predicates in \( C \) are safe; predicates in \( C_j \) are safe from condition (1)(ii). Thus, the given magic rule can only generate a finite number of tuples.

We have shown that all predicates in \( P_{\text{mg-mod.} C} \) and their magic predicates are safe (and therefore all strongly safe). This completes the proof of Theorem 7.3.

Example 7.1. Consider the following program \( P \):

\[
\begin{align*}
p(X) & : = f(X, U), q(U, Y), p(Y). \\
p(X) & : = b(X). \\
q(X, Y) & : = q(X, Z), q(Z, Y). \\
q(X, Y) & : = c(X, Y). \\
p(5) & .
\end{align*}
\]

where \( b \) and \( c \) are finite base relations and \( f \) is an infinite base relation in which the FCs \( f_1 \rightarrow f_2 \) and \( f_2 \rightarrow f_1 \) hold. Let this generate the adjoined program \( P_{\text{mg}} \) (by an appropriate choice of sips):

\[
\begin{align*}
p^1(X) & : = f(X, U), q^1(U, Y), p^1(Y). \\
p^2(X) & : = b(X). \\
q^1(X, Y) & : = q^1(X, Z), q^1(Z, Y). \\
q^2(X, Y) & : = c(X, Y). \\
p^5 & .
\end{align*}
\]

There are two cliques, \( C_1 \) and \( C_2 \), containing \( q^1 \) and \( p^2 \), respectively. The corresponding magic program, \( P_{\text{mg}} \) (which is the same as \( P_{\text{mg-mod.} C_2} \)), is:

\[
\begin{align*}
p^1(X) & : = magic \_ p^1(X), f(X, U), q^1(U, Y), p^1(Y). \\
p^2(X) & : = magic \_ p^2(X), b(X). \\
q^1(X, Y) & : = magic \_ q^1(X), q^1(X, Z), q^1(Z, Y). \\
q^2(X, Y) & : = magic \_ q^2(X), c(X, Y). \\
\end{align*}
\]

We have not shown the see fact \( magic \_ p^5(5) \), since it is not used in the analysis. Of course, we assume that it is added before evaluating the program.

The following rules for \( q^f \) are also included in \( P_{\text{mg}} \):  
- \( magic \_ q^f(U) : = magic \_ p^f(X), f(X, U), q^f(U, Y) \)
- \( magic \_ q^f(Z) : = magic \_ p^f(X), f(X, U), q^f(Z, Y) \)

Since this is a Datalog program, it is strongly safe. If we consider the rules defining \( magic \_ q^f \) generated from clique \( C_1 \) (only rule 7)—note that we ignore rule 6 since it is not generated from \( C_1 \), they are strongly safe if we replace \( q^f \) by a finite base relation. (In this example, they are strongly safe even without this replacement.) Thus \( q^f \) and \( magic \_ q^f \)
are both safe in $P_{mg,mod,C}$ (rules 3’, 4’, 7’ and the rules for $q^f$), and, since these are the only derived predicates in $P_{mg,mod,C}$ (other than $q^f$, which is already known to be safe), they are also strongly safe. We have shown that condition (1) of Theorem 7.3 holds. Therefore, in order to show the strong safety of $P_{mg,mod,C}$ (i.e., $P_{mg}$), we only need to establish condition (2) of the theorem. For this purpose, we obtain a program $P'$ by modifying $P_{mg,mod,C}$ as follows: (1) we delete rule 6 (which was obtained from rule 1 in clique C2), and (2) we replace occurrence of $p^b$ in rules defining $p^b$ by a new finite base relation $d$. This gives us the following rules for $P'$:

1. $p^b(X) := \text{magic} \cdot p^b(X), f(X, U), d(U, Y), p^b(Y)$.
2. $p^b(X) := \text{magic} \cdot p^b(X), b(X)$.
3. $q^f(X, Y) := \text{magic} \cdot q^f(X), q^d(X, Z), q^d(Z, Y)$.
4. $q^f(X, Y) := \text{magic} \cdot q^f(X), c(X, Y)$.
5. $\text{magic} \cdot p^b(X), f(X, U), d(U, Y)$.
6. $\text{magic} \cdot q^d(Z) := \text{magic} \cdot p^b(X), q^d(X, Z)$.

Rules 3’, 4’, and 7’ constitute (part of) program $P_{mg,mod,C}$, as we saw above, they generate only a finite number of tuples for $q^f$ and $\text{magic} \cdot q^f$. Thus, we need to consider the program consisting of rules 1’, 2’, and 5’ and show that it is strongly safe. The predicate $\text{magic} \cdot p^b$ is safe since the (only) head variable appears in a d atom in rule 5’ (the only rule defining $\text{magic} \cdot p^b$); the safety of $p^b$ follows since in both rules 1’ and 2’ (the only rules defining $p^b$), the (only) head variable appears in a $\text{magic} \cdot p^b$ atom.

The use of Theorem 7.3 permitted us to analyze clique C2 relatively independently of the details of clique C1. We note that we did not explicitly consider rule 6, which defines $\text{magic} \cdot q^f$. The safety of this rule comes for “free.” (It is a consequence of the claim that is stated and established in the proof of the theorem.) Indeed, algorithm check_clique would not show safety of $\text{magic} \cdot p^b$ if it were directly applied.

We observed that $P_{mg,mod,C}$ correctly computes all answers to a query of the form $q^f(\bar{v}, \bar{X})$, where $q^f$ is in C1, given a seed fact $\text{magic} \cdot q^f(\bar{v})$. Given the additional condition that $q^f(\bar{v}, \bar{X})$ was safe in $P_{mg}$, we used this observation to show that $q^f$ could be treated as a finite base relation for purposes of safety analysis. If $q^f(\bar{v}, \bar{X})$ cannot be shown to be safe in $P_{mg}$, the following theorem offers an alternative approach.

**Theorem 7.4.** Let the topologically sorted set of cliques over $P_{mg}$ be represented as a dag. Let $C$ be a clique all of whose children are leaves. Given:

1. For all Ci such that Ci is a child of C, for all $q^n$ in Ci, (i) $q^n$ is strongly safe in $P_{mg,mod,C}$.

2. For all Ci such that Ci is a child of C, for all $q^n$ in Ci, replace every occurrence of $q^n$ in the body of a rule of $P_{mg,mod,C}$ that is not also in $P_{mg,mod,C}$ by an infinite base predicate in which the bound (b) argument positions finitely constrain the free (f) argument positions. Further, delete all rules defining $\text{magic} \cdot q^n$ that are obtained from such occurrences of $q^n$. The program $P'$ obtained from $P_{mg,mod,C}$ by making these changes is strongly safe.

**Then:** $P_{mg,mod,C}$ is strongly safe.

**Proof.** The proof follows the same lines as the proof of Theorem 7.3. We may treat $P_{mg,mod,C}$ as a subprogram that is used to compute the answer to a subquery $q^n(\bar{v}, \bar{X})$, where $q^n$ is in C1. Condition (1) and Lemma 7.2 ensure that this subprogram will compute all the answers and terminate, for a given vector of values $\bar{v}$ (the seed). This vector is passed to the subprogram in the form of a magic fact computed using the magic rule in $P_{mg,mod,C}$ generated from the given occurrence of $q^n$. Since we cannot establish that the union of the sets of answers for all seeds is finite, we cannot view this occurrence of $q^n$ as a finite base relation. However, the set of answers for a given seed is finite and is effectively computable. This is modeled by an infinite base relation with the same bound and free arguments in which the bound arguments functionally constrain the free arguments. The remainder of the proof is identical to that of Theorem 7.3.

**Example 7.2.** Consider the program $P$:

$p(X) := q(X, Y), p(Y)$.

$q(X, Y) := b(X)$.

$q(\bar{v}, \bar{X}) := f(X, Y)$.

$p(\bar{v})$.

where $b$ is a finite base relation and $f$ is an infinite base relation in which the first argument determines the second. The adorned program is

1. $p^b(X) := q^d(X, Y), p^b(Y)$.
2. $p^b(X) := b(X)$.
3. $q^d(X, Y) := f(X, Y)$.
4. $p^b(\bar{v})$.

This has two cliques, C1 and C2, containing $q^d$ and $p^d$, respectively. The magic program $P_{mg}$ is

1. $p^b(X) := \text{magic} \cdot p^b(X), q^d(X, Y), p^b(Y)$.
2. $p^b(X) := \text{magic} \cdot p^b(X), b(X)$.
3. $q^d(X, Y) := \text{magic} \cdot q^d(X), f(X, Y)$.
4. $\text{magic} \cdot p^b(\bar{v})$.
5. $\text{magic} \cdot p^b(Y) := \text{magic} \cdot p^b(X), q^d(X, Y)$.
6. $\text{magic} \cdot q^d(X) := \text{magic} \cdot p^b(X)$. 

File: 571J 138323 . By:CV . Date:19:01:00 . Time:16:07 LOP8M. V8.0. Page 01:01
Codes: 6572 Signs: 4596 . Length: 56 pic 0 pts, 236 mm
The rule defining \( q'' \) is

\[
q''(X, Y) : = f(X, Y).
\]

This is clearly unsafe, and so Theorem 7.3 does not apply. However, consider \( P_{\text{mg-mod.C1}} \), which consists of just rule \( 3' \). The predicate \( \text{magic}_g \) is safe in \( P_{\text{mg-mod.C1}} \) trivially since there is no rule defining it; all magic facts must be provided as "seeds" (by applying rule 6, which is not in \( P_{\text{mg-mod.C1}} \)). We see that \( q'' \) is strongly safe in \( P_{\text{mg-mod.C1}} \), since \( \text{magic}_g \) is safe and \( Y \) is finitely constrained by \( X \) in \( f \). Thus, the first hypothesis of Theorem 7.4 is satisfied. If we replace \( q'' \) in \( C_2 \) by an infinite base relation whose first argument finitely constrains the second, we obtain

1'. \( p^h(X) : = \text{magic}_g p^h(X), g(X, Y), p^h(Y) \).
2'. \( p^h(X) : = \text{magic}_g p^h(X), b(X) \).
4'. \( \text{magic}_g p^h(5) \).
5'. \( \text{magic}_g p^h(Y) : = \text{magic}_g p^h(Y), g(X, Y) \).

where \( g \) is an infinite base relation with the FC \( g_1 \rightarrow g_2 \). It turns out that rule 5' is unsafe, and therefore the second hypothesis of Theorem 7.4 is not satisfied. So we cannot use Theorem 7.4 to show that the program is safe (which is indeed unsafe).

On the other hand, if the first rule in the given program had an additional body atom \( d(Y) \), where \( d \) was a finite base predicate, the analysis would have proceeded similarly and established that \( \text{magic}_g p^h \), and also \( p^h \), was safe. In this case, both hypotheses of Theorem 7.4 would be satisfied, and thus the program would be declared safe.

We note that the replacement of a derived predicate by an infinite predicate in the safety analysis could result in loss of information that is essential to showing strong safety. Suppose \( q \) were defined by the following rule:

\[
q(X, Y) : = f(X, Y), b(Y).
\]

Then, \( q'' \) would still unsafe we would have to use Theorem 7.4, and we would arrive at the same set of rules for \( C_2 \) (rules 1', 2', 4', 5'). However, in replacing \( q'' \) by \( g \), we have lost the information that the second argument is bounded (since it can only receive values from the finite base relation \( b \)). Thus, since rule 5' is unsafe, Theorem 7.4 cannot be used to show \( P_{\text{mg}} \) to be safe. However, \( P_{\text{mg}} \) is in fact strongly safe.

8. CONCLUSIONS

This paper makes two main contributions. The first is an algorithm (\text{check}_\text{clique}) for checking safety and effective computability in a bottom-up model of execution of an arbitrary recursive clique using monotonicity and finiteness constraints. This algorithm provides the central component of a framework for testing programs. The framework makes the testing of large programs tractable by organizing programs into cliques and permitting a clique by clique analysis. The check\_clique algorithm is parametrized by procedures that are assumed to be available, such as check\_prop, tag\_prop, and auxiliary procedures for inferring boundedness and monotonicity constraints. The procedures check\_prop and tag\_prop represent abstractions of operations that are implicitly in any safety analysis involving monotonicity constraints, and in this paper they are isolated any specified rigorously. By providing more sophisticated versions of these procedures, the results presented here can be used to show safety and effective computability of a larger class of programs.

The second contribution is a methodology for analyzing the safety of programs produced by the Generalized Magic transformation that takes advantage of the special properties of the transformation to avoid directly examining the (more complicated) clique structure of the rewritten program.

Further work is needed in inferring monotonicity constraints, possibly interacting with the programmer. There are two aspects to this problem. In general, if we cannot show a clique to be strongly safe, it is not clear whether additional constraints will help us to do so, and if so, exactly what these additional constraints are. Also, once we identify a potentially useful constraint, we need to determine whether this constraint holds. Relevant results are reported in [14, 6]. We also need to explore well-founded orderings that can be used in testing the safety of cliques, in order to find orderings that are easy to test, and widely applicable. Finally, we emphasize that the framework presented here can be used to test safety and effective computability under various sideways information passing strategies by suitably rewriting the program and then analyzing the rewritten program.

REFERENCES


