A Framework for Testing Safety and Effective Computability of Extended Datalog[†]

Ravi Krishnamurthy

Raghu Ramakrishnan

Oded Shmueli

Computer Sciences Technical Report #774

June 1988

[†]A shorter version of this paper appeared in Proc. SIGMOD, 1988.

A Framework for Testing Safety and Effective Computability of Extended Datalog

Ravi Krishnamurthy (1)
Raghu Ramakrishnan (2)
Oded Shmueli (1,3)

ABSTRACT

This paper presents a methodology for testing a general 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 bottom-up fixpoint computation. We also approximate 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 considered. 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. 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 algorithm *check_clique*. This work provides a framework for testing safety and effective computability of recursive programs, and is based on a clique by clique analysis. The results reported here form the basis of the safety testing for the LDL language, being implemented at MCC.

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:

$anc(X, Y) :- par(X, Y)$
$anc(X, Y) :- par(X, Z), anc(Z, Y)$

and let the query be

$Query: anc(john, Y)$?

Assume that a database contains a parenthood relation *par*. Then the program defines a derived relation describing ancestors, and the query asks for the ancestors of *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 which 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 same program is written as follows:

$anc(X, Y) :- par(X, Y)$
$anc(X, Y) :- anc(X, Z), par(Z, Y)$

The second problem with Prolog is that its evaluation strategy always computes joins by a nested loop join technique which is inefficient in the presence of a large number of facts [Bancilhon and Ramakrishnan 87].

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 [Lloyd 84]. 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

(1) MCC, Austin.
(2) University of Wisconsin-Madison. Work partly done while visiting MCC.
(3) Technion, Israel.

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 [Bancilhon and Ramakrishnan 86]. The main thrust of these strategies is to improve efficiency by restricting the computation to tuples that are related to the query. Since these bottom-up strategies promise to be an efficient approach to 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 still require 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 [Shmueli 87, Gaifman 86]. 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 questions of safety and termination are important questions for general logic programs, they are particularly of interest for programs expressing database queries. We address these questions in this paper. We consider programs which 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 evaluable functions such as arithmetic operations. We use the name *Extended Datalog* to denote Datalog with this extension.

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. 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 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.

Our framework for testing safety and effective computability is based on a simple bottom-up model of execution. We assume that the given program has already been rewritten, if necessary, according to the desired sideways information passing strategies. Thus, in the safety testing phase, we only check the bottom-up fixpoint computation of the given program, assuming that it will not be further transformed or otherwise optimized. 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.

The *capture rules* framework proposed in [Ullman 85] also addressed the issues of safety and effective computability: if a query can be "captured" using some known capture rules, then there is an algorithm which computes all answers. 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 [Ullman 85, Sagiv and Ullman 85, Afrati et al. 86, Ullman and Van Gelder 85]. On the other hand, several of the examples presented here cannot be handled using the previously known capture rules.

The rest of this paper is organized as follows. We present some 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 problems involved in testing recursive cliques for safety. In section 5, we develop algorithm *check_clique* and prove its correctness. We discuss related work in section 6, and present our conclusions in section 7.

2. Definitions

In this section, we review a number of basic concepts from the literature.

A Horn Clause is a rule of the form $p(t)$:- $q1(t1), q2(t2), \ldots, qn(tn)$, where $p(t)$ and the $qi(ti)$s are *literals*. A *literal* 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. An example of a function symbol is

the cons operator in Lisp. The literal p is called the *head* of the rule, and the rest of the rule is called the *body*. The semantics associated with a rule is that $p(t)$ is true if all the $qi(ti)$s are true, i.e., the tuple t is in the relation p if tuple t_i is in relation q_i for all i. It is well known [Lloyd 84] 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). A *query* is a rule without a head. We will assume here that a query is a single predicate occurrence, and that all its arguments are distinct variables. † The set of answers to a query $q(\bar{X})$? is the set of all facts $q(\bar{c})$ in the least fixpoint.

We only consider programs in which all arguments are variables or constants. Thus, terms constructed using function symbols are not allowed. We approximate such terms by allowing infinite base relations. 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.

```
concat([X|Y],Z,[X|U]) :- concat(Y,Z,U).
concat([],Z,Z) :-.
```

We approximate it as follows in Extended Datalog:

```
concat(V,Z,W) :- concat(Y,Z,U), h(X,Y,V), h(X,U,W).
concat([],Z,Z) :- .
```

The predicate h is an infinite base predicate corresponding to the function symbol |. Conceptually, it contains all triples (element, list1, list2) such that list2 is the term [element|list1]. []

We partition the given Extended Datalog program into a set of *facts* which 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 which appears in the head of a rule. Without loss of generality, we assume that the EDB and IDB are disjoint sets of predicates. 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 (EDB, IDB, IC).

A *ground term* is a term containing no variables. The facts in the EDB are just rules with an empty right hand side in which all arguments (of the head predicate) are ground terms. The EDB may contain predicates which 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.

We use the convention that infinite base predicates are denoted by f, g, h, ... , finite base predicates by a, b, ... and derived predicates by p, q, ... Argument places are referred to by the predicate name subscripted by the place number. For instance, p_i refers to the ith argument of predicate p. Variables are denoted by upper case letters and constants are denoted by numerals.

Programs may contain predicates which 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) :- p(Y), X = Y+1
```

The presence of such a rule would lead us to infer that this rule could generate an infinite number of tuples for p by repeated applications of this rule. However, since this is the only rule defining p, it is empty. A polynomial time algorithm is presented in [Ramakrishnan et al. 87] for identifying predicates, such as p, which are empty for all instances of the EDB. Without loss of generality, we will henceforth assume that

† 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 which is evaluated bottom-up. This issue is discussed further at the end of section 5, and in section 6.

this algorithm has been applied to the given program to remove all rules defining such predicates.

We now consider the issue of *safety*. A given query is *safe* if it has a finite set of answers for all instances of the EDB which satisfy all integrity constraints. Thus, a query is *unsafe* if there is some instance of the EDB which satisfies the integrity constraints and is such that the query has an infinite set of answers. The integrity constraints we consider in this paper are *finiteness constraints* and *monotonicity constraints*.

A *finiteness constraint* (FC) over a base predicate r is of the form X --> Y, where X and Y are sets of arguments. 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[Y] | s[X] = t[X] }, is finite. Note that this definition is strictly weaker than the traditional definition of a functional dependency; it holds trivially for all finite predicates (and in particular, for all finite EDB predicates).

Example 2.3

Consider the predicate f(X,Y). If we define f(X,Y) by Y = 2*X, then we have the finiteness constraints f_1 --> f_2 and f_2 --> f_1. If we define g(X,Y) by X < 0 and Y > 0, there is no finiteness constraint between the arguments of 'g'. If we define h(X,Y) by X > 0 and Y = 0 or 5, we have the finiteness constraint h_1 --> h_2. []

Let r_i and r_j be arguments of predicate r. A *monotonicity constraint* is a couple $r_i > r_j$. The constraint $r_i > r_j$ holds in an instance of r if and only if the value in the ith argument is strictly greater than the value in the jth argument in every tuple. We may also specify a monotonicity constraint $r_i >$ c (resp. $r_i <$ c) where 'c' is a constant. This constraint holds in an instance of r if and only if the value in the ith argument is strictly greater (lesser than) than the constant 'c'. Of course, this 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 is discussed further after Example 4.2.)

Example 2.4

Consider the predicate f(X,Y). If we define f(X,Y) by Y = 2*X, then we have the finiteness constraints f_1 --> f_2 and f_2 --> f_1. If we define f(X,Y) by Y = 2*X, X > 0, Y > 0, we have the finiteness constraints f_1 --> f_2 and f_2 --> f_1, and the monotonicity constraint $f_2 > f_1$. If we define f(X,Y) by X < 0 and Y > 0, there is no finiteness constraint between the arguments of 'f', but the monotonicity constraint $f_2 > f_1$ holds. If we define f(X,Y) by X > 0 and Y = 0 or 5, we have the finiteness constraint f_1 --> f_2, but there is no monotonicity constraint relating X and Y. []

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 *<- q. p is a *recursive* predicate if p *<- p. Two predicates p and q are *mutually recursive* if p *<- q and q *<- p. A *clique* is a maximal set of mutually recursive predicates. We assume that each non-recursive predicate forms a singleton clique.

We begin by introducing several concepts and definitions. For each fact that belongs to a derived predicate, there exists a finite *derivation tree*, which describes how it is derived from base facts using rules of the program. Let $p(c)$ be a fact in the derived predicate p. Then the tree has $p(c)$ at its root, the leaves are base facts, and each internal node is labeled by a fact, and by a rule which generates this fact from the facts labeling its children. A base fact may be viewed as a derivation tree of height one.

In order to reason with monotonicity constraints, we need the notion of an *argument mapping*. Our definition is similar to the definitions presented in [Afrati et al. 86, Ramakrishnan et al. 87]. Let p be the head and q be a derived literal occurrence in the body of a rule r. An argument mapping (p, q, r) is a graph with the set of nodes being the argument places of p and q.† For each rule, we obtain an argument mapping between the head literal and each derived literal occurrence in the body. We draw an undirected *edge* between two nodes if the same variable occurs in the corresponding argument positions. We draw an *arc* from one node, say $n1$, to another, say $n2$ if variables X and Y appear in the corresponding argument

† Without loss of generality, we assume that each literal 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 literal in the rule, together, as the literal name. We also assume that all variable names are unique. We note that the third component of an argument mapping is redundant, but we include it for ease of exposition.

positions, and we can infer that X > Y from the monotonicity constraints in this rule.

Argument mappings ($p, q, r1$) and ($q, m, r2$) can be composed to yield a composite mapping ($p, q, r1$).($q, m, r2$) by merging the corresponding nodes in q. In particular, the composite mapping ($p, q1, r$).($q1, q2, r1$) ... (qn, p, rn) represents a *cyclic composite argument mapping*, and we can complete the cycle by joining the corresponding nodes in the two instances of p with edges. Further, if $qi <> p$, i = 1 to n, and $qi <> qj$, i <> j, then this is a *simple cycle*.

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 in a derivation tree induces a rule sequence. In fact, every path in a derivation tree induces a composite argument mapping. (In addition to the rule sequence, we must consider, at each node in the path, which literal in the rule body was expanded.) We thus often speak of the composite mapping *corresponding* to a path in a derivation tree. Also, since many paths in derivation trees may induce the same composite mapping, we include all such paths in the set of paths *corresponding* to the composite mapping.

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*, that is, whether the evaluation strategy computes all answers and terminates, and every intermediate relation is of finite size. Effective computability implies safety, but the converse is not true. In order to discuss effective computability, we define a stronger notion of safety.

A predicate p is *strongly safe* if every derived predicate q such that $p \leftarrow q$ 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 which appears in the body is strongly safe.

We now present an overview of our approach to testing a program. Clique C1 is a *child* (resp. *descendant*), of clique C2 if $p \leftarrow q$ (resp. * <–) for some p in C2 and q in C1. We denote this as C2 ≤ C1 (resp. C2 *≤ C1). We observe that *≤ is a partial order. A clique C is a *leaf* clique if there is no clique C1 such that C ≤ C1. 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 belong to it.

Let C1 be a leaf clique, and let q be a predicate in C1 which is used to define some predicate in another clique C2. After analyzing clique C1, 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 C1 which are used to define predicates outside C1, we remove C1 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. Our *model of execution* is the following: We assume that a given program P is first transformed to P^{mg}, which is then evaluated bottom-up. Thus, the rules are repeatedly applied (in any order) until no new tuples are produced by the application of any rule. A rule is *applied* by taking the join of (the current instances 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 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, Let $\{f_i, \ldots, f_j\}$ be the set of argument positions such that the variables in them appear to the left of this occurrence of f in the body of the rule. Then, these arguments must determine all arguments of f through the FCs given to hold over f.
3. Consider an infinite base relation f with an FC $f_i, \cdots f_j \rightarrow f_k$. Given an assignment of constants to argument positions $f_i, \cdots f_j$, we can compute the (finite) set of values appearing in the argument position f_k in a finite amount of time.

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 is also finite.

Lemma 3.1:

Each application of a rule produces only a finite number of tuples for the head predicate and is effectively computable, given the three assumptions about programs.

Proof:

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 at each stage, although it may be unbounded). 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 these arguments determine all the other arguments, by assumption 2. Given a vector of values for the bound arguments, the corresponding vectors of values for the other arguments are effectively computable by 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, thus completing our proof. []

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 constructed finite intermediate relations. 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). From Lemma 3.1, given our three assumptions about programs, 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 only need to show that every derived predicate is safe, and that the computation terminates after a finite number of rule applications.

We have the following theorem characterizing strongly safe queries.

Theorem 3.2:

If a query is strongly safe, then it is effectively computable according to the above model of execution, given the three assumptions about programs.

Proof:

By Lemma 3.1, each rule application is effectively computable, given the three assumptions about programs. Thus, we only need to show that the computation halts after a finite number of rule applications.

For every predicate that is used to define the query, strong safety implies that it is safe 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 halt. (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 recursion.

Consider a property *prop*, defined over sequences, and a predicate p. Consider a path in a derivation tree. We say that this path satisfies property *prop* over predicate p if the sequence of facts of the form $p()$ on this path satisfies property *prop*. We also say that the path satisfies (p, *prop*), for brevity. For example, *prop* could be defined over a sequence of k-ary facts to mean that for any pair of facts, if fact 1 precedes fact 2 in the sequence, the ith argument of fact 1 is < the ith argument of fact 2, for some i. If the ith argument in any fact is an integer, < could simply be the usual ordering defined over integers. As another example, *prop* could be the property that the ith argument in any fact in a sequence of k-ary facts should be a

positive integer less than 10. For a third example, *prop* could be defined as the conjunction of the previous two properties.

A property is *well-founded* if the following holds. Let S be a sequence which satisfies the property, and let D be the domain of elements in S. Then: (1) We can define an ordering $\ll$ such that $ei \ll ei+1$, for all adjacent elements ei, $ei+1$ in the sequence S. (2) For any $c1$, $c2$ in D, if $c1 \ll c2$, then there are only a finite number of values c in D such that $c1 \ll c \ll c2$. (3) There is an element c in D such that $c \ll c1$ for all $c1$ in D.

A large class of well-founded properties can be specified by simply specifying an argument position, an ordering, and a bound. 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, present no difficulties.

Given a rule sequence, we say that the rule sequence satisfies $(p, prop)$ if every path in the set of paths corresponding to this rule sequence satisfies $(p, prop)$.

We say that a variable X is *bounded* if it can only take on values from a finite domain. An argument position in a literal occurrence is *bounded* if it contains a constant or the variable in it is bounded. Similarly, we say that 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 that is bounded above (below).

Consider a cyclic (composite) argument mapping $(p, q1, r).(q1, q2, r1)$ ... (qn, p, rn). The corresponding rule sequence is $r, r1, \ldots rn$. 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 applying the sequence of rules $rn, \ldots, r1, r$ assigns *increasing* (resp. *decreasing*) values to the nodes on the cycle in the argument mapping. (Clearly, such a cycle must contain at least one arc, and it cannot contain two oppositely directed arcs. Our choice of directions for *increasing* and *decreasing* cycles reflects our bottom-up application of rules.)

Note that every path corresponding to an increasing (resp. decreasing) cycle satisfies a simple increasing (resp. decreasing) property. Further, if the cycle is bounded, then the property is well-founded. Consequently, any single rule clique is safe if a well-founded property is associated with it. 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 C1 and C2, containing q and p respectively. Consider the argument mapping corresponding to the rule cycle (1,1) in C2: p(X,Y), p(U,V). There is an arc from U to X, and one from V to Y. Thus, we have an increasing cycle and a decreasing cycle in this argument mapping. Further, X is bounded since it appears in a 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 C2 is therefore strongly safe. A similar analysis shows C1 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 C2 depend on C1. However, C1 and C2 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(U,V) :- p(U,V)

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 rule cycles considered before, there is the rule cycle (5,6), whose argument mapping is (p(X,Y), q(Y,X), p(Y,X)). The exchange of arguments enables repeated applications of this rule cycle, in conjunction with rule cycles (1,1) and (2,2), to produce an infinite number of tuples. []

Consider a vector of m arguments. It is possible to identify properties that characterize the vector (or some sub-vector) 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. Consider either argument position by itself does not allow us to conclude that this program is safe since neither argument by itself is bounded. []

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 the powerset 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 in an argument mapping.

The purpose of this section has been to give the reader some intuition into 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 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 that 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 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 the height of all derivation trees is bounded for all EDBs, then every predicate is strongly safe.

Proof:

From Lemma 3.1, each application of a rule produces only a finite number of tuples. If the height of all derivation trees is bounded, then the maximum number of rule applications involved in producing any fact is bounded. Thus, the number of possible facts which can be produced by rule applications is also bounded. []

We present an algorithm to test 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 cyclic mapping $(p, \_, \_), \ldots, (\_, p, \_)$ 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 $prop_p$, which is well-founded with respect to this argument mapping. Having done this for all mappings, we verify that the interaction of rule cycles does not invalidate the assumptions made in checking them individually. For example, if $(p, \_, \_), \ldots, (\_, p, \_)$ 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 *check* and *tag_prop*. We now specify them formally.

check $(C, prop)$:

> Let cycle C = $(p, q_1, r_1), (q_1, q_2, r_2), \ldots, (q_n, p, r_n)$. The procedure *check* $(C, prop)$ returns *true* only if the following holds: Every path P = $(p(), q_1(), q_2(), \ldots, q_n(), p())$ in a derivation tree satisfies $(p, prop)$.

tag_prop $(C, prop, q_j, PROP_j)$:

> Let cycle C = $(p, q_1, r_1), (q_1, q_2, r_2), \ldots, (q_n, p, r_n)$. The procedure *tag_prop* $(C, prop, q_j, prop_j)$ succeeds with $PROP_j = prop_j$, for j = 1 to n; where $prop_1, prop_2, \ldots, prop_n$ are some n properties such that the following holds:
>
> Let Qi = $(q_i(), \ldots, q_i())$, i = 1 to n, be any n paths such that Qi satisfies $(q_i, prop_i)$, i = 1 to n. Let P = $(p(), q_1(), q_2(), \ldots, q_n(), p())$ be any path in a derivation tree which satisfies $(p, prop)$. Then the path $(p(), Q_1(), Q_2(), \ldots, Q_n(), p())$ also satisfies $(p, prop)$.

For example, if *prop* is the property that some argument of p is increasing then there is some increasing cycle in the argument mapping C which contains occurrences of this argument of p. If some argument of q_j appeared in this cycle, then $prop_j$ is the property that this argument of q_j should increase or remain the same. Note that this is *not* a well-founded property.

We present a recursive procedure *check_clique* which examines the interactions between rule cycles. We prove that if this procedure returns *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 DB. Thus, the number of derivation trees for facts in DB is bounded. (This implies that every predicate is safe, and it immediately follows that every predicate is strongly safe as well. Effective computability then follows from Theorem 3.2.)

We note that a given rule cycle may be used to expand any predicate which occurs on it, and so if the cycle contains two predicates p and q, the corresponding argument mapping can be denoted as either $(p, \_, \_)$, ... , $(\_, p, \_)$ or $(q, \_, \_)$, ... , $(\_, q, \_)$. While 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 cycles of the form $(p, \ldots, p)$ we are assured that indeed all simple rule cycles containing p are considered.

The following procedure checks an arbitrary clique for strong safety.

proc check_clique: boolean /* The top-level procedure */

1. For each predicate *pi* in the given clique, associate a well-founded property $prop_i$.

2. FLAG := true; i = 1;
   while FLAG and i ≤ n **do** /* n is the number of predicates in C */
   FLAG := FLAG & verify(*pi*, {*pi*}, $prop_i$);
   i := i+1
   od.

3. return FLAG.

end. /* *check_clique* */

proc verify(*p*, IGNORE, prop): **boolean**

/* Does the detailed work. Checks if *prop* is a well-founded property for each simple composite argument mapping $(p, \_, \_)$, ... , $(\_, p, \_)$. 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 subprograms, *check* and *tag_prop* are given. */

1. Let {*C* 1, ... , *Cm*} be the set of simple cyclic argument mappings of the form $(p, \_, \_)$, ... , $(\_, p, \_)$. We first verify that *prop* is a well-founded property for each of these cycles considered separately.

```
FLAG := true;
for all i in {1, ... , m} do
  /* Ci' is the set of predicates on Ci = (p, _, _), ... , (_, p, _), except for p. */
  if Ci' ∩ IGNORE = {} then
    /* check verifies that (p, prop) is preserved by the sequence of predicates
       on Ci if they are not expanded further */
    FLAG := FLAG & check(Ci, prop);
    Let q 1, ... , qk be the recursive predicates in Ci'.
    for all j in {1, ... , k} do
      tag_prop(Ci, prop, qj, prop_j);
    od
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, ... , m} do
  if Ci' ∩ IGNORE = {} then
    Let q1, ... , qk be the recursive predicates in Ci'.
    for all j in {1, ... , k) do
      IGNOREQ := IGNORE ∪ {qj};
      /* prop_j is the property with which qj is "tagged" by tag_prop(Ci, prop, qj, prop_j) in Step 1. */
      FLAG := FLAG & verify(qj, IGNOREQ, prop_j)
    od
od
```

3. return (FLAG).

end /* *verify* */

We now present our main theorem.

Theorem 5.2:

If *check_clique* 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:

We prove that all paths to a leaf in a derivation tree are bounded in length if *check_clique* returns *true*. We argue by induction on the number of recursive predicates which appear on such a path. Let us define RSym(path) to be the set of recursive predicate symbols p such that there is a fact $p(c)$ on the path.

Induction Hypothesis: There exists a constant $L_{(C,DB,k)}$ such that for every path, if ||RSym(path)|| < k, then the length of the path is $< L_{(C,DB,k)}$.

Basis: Consider a path with only one recursive predicate, say p. Let the path be $q_1(), \ldots, q_n(), p(), \ldots, p()$, where the q_i are non-recursive predicates. The segment $p(), \ldots, p()$ must correspond to applications of simple argument mappings of the form $(p, \_, \_), \ldots, (\_, p, \_)$. Procedure *check_clique* ensured that each application of such an argument mapping preserved the well-founded property associated with p. (This was done through the call to *verify* in *check_clique*, and the call to *check* 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 well-founded property 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: $q_1(), \ldots, q_n()$. From Lemma 3.1, it follows that the set of possible values c is finite. Given the well-founded property associated with the segment $p(), \ldots, p()$, this ensures that the length of the path is bounded by some constant $L_{(C,DB,1)}$ (which depends on the clique C and values in the database DB).

Inductive Step: Suppose the inductive hypothesis holds. Consider a path on which k recursive predicates occur. If this path is unbounded, some predicates must appear an unbounded number of times. Let p be the first such predicate. Consider the segment of the path between any two adjacent occurrences of p. We will prove that this segment preserves the well-founded property associated with p. That is, in going from an occurrence of p to an adjacent occurrence of p, the value of the well-founded property uniformly increases (or decreases), and since these values are bounded and initially finite by the same argument as for the basis case, the number of occurrences of p is bounded. Thus, by contradiction, we establish that each path containing k recursive predicates must be bounded.

It remains to show that each segment between two adjacent occurrences of p preserves the well-founded property for p (Lemma 5.3). By our induction hypothesis, since this segment cannot contain p and therefore contains fewer than k recursive predicates, it is bounded in length. We thus complete our proof of Theorem 5.2 by establishing Lemma 5.3. []

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 path in derivation tree can be produced by applications of this transformation (Lemma 5.4). We then prove that every path $(p(), \ldots, p())$ so produced preserves $(p, prop)$, where *prop* is the property associated with p (Lemma 5.5). To prove the latter claim, we define a structure called a *transformation tree* which 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*. (The role of the set IGNORE in *check_clique* is addressed in Lemma 5.5.)

Lemma 5.3:

If *check_clique* 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 begin by defining a transformation T.

The transformation T consists of choosing a node, say q, from the input segment $(p, \ldots, p)$, where q is distinct from p, or q is the occurrence of p at the right end of the sequence, and replacing it with a simple cycle $(q, \ldots, q)$. There are two restrictions:

1. Once we choose a node, no node to the left (that is, higher in the derivation tree) is subsequently chosen for replacement.
2. Consider the node q chosen for expansion. If it was produced from the initial simple cycle $(p, \ldots, p)$ by first replacing a node q_1, then replacing a node q_2 from the simple cycle used to replace q_1, and so on, before finally replacing q_m to produce q, then the simple cycle used to replace q should not contain any of the predicates $p, q_1, q_2, \ldots, q_m$, except at the endpoints. (Of course, the endpoints of this simple cycle are nodes q, and it is possible that q is identical to q_m.)

We prove Lemma 5.3 in two steps. We first show that each bounded length segment $(p, \ldots, p)$ can be generated by starting with some simple cycle $(p, \ldots, p)$ and repeatedly applying T (Lemma 5.4). We then show that any segment $(p, \ldots, p)$ produced by applying T repeatedly to a simple cycle $(p, \ldots, p)$ preserves the well-founded property associated with p (Lemma 5.5). This concludes the proof of Lemma 5.3. []

Lemma 5.4:

Each bounded segment $(p, \ldots, p)$ in a path of a derivation tree can be generated by starting with a simple cycle $(p, \ldots, p)$ and repeatedly applying transformation T.

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: If the length is 1 (excluding the endpoints), this must be a simple cycle, and the claim holds trivially. (If there is no simple cycle of length 1, the basis is on the length of the shortest simple cycle.)

Inductive Step: Consider a segment $(p, \ldots, p)$ of length k. Scanning from right to left, let $q1$ be the first predicate which appears again later in the segment. Consider the subsequence from the leftmost occurrence of $q1$ to the rightmost occurrence of $q1$ in the given segment, and denote this sequence as $Q1$. Let the next predicate (following the rightmost occurrence of $q1$) which repeats be $q2$. Define the subsequence $Q2$ as before, and so on for all repeating predicates. It is easy to see that the given segment $(p, \ldots, p)$ can be represented as $(p, N1, Q1, N2, Q2, \ldots, p)$, where Ni is a subsequence of predicates that do not appear anywhere else in the given segment $(p, \ldots, p)$. Further, it follows that there must be a simple cycle $(p, q1, N1, q2, N2, \ldots, p)$. (By construction, no node in any segment Ni appears anywhere else, and each qi appears only once. Thus, this is a simple cycle.) Since $Q1$ denotes a segment $(q1, \ldots, q1)$ of length less than k, by the inductive hypothesis, it can be generated by starting with $q1$ and repeatedly applying transformation T. We can similarly generate the segments Qi in order, by expanding the corresponding qi. In doing so, no nodes to the left of qi are replaced again. Thus, we have shown how the given segment of length k can be generated by repeated applications of T (which obey the two restrictions in the definition of T). This concludes the proof of Lemma 5.4. []

Lemma 5.5:

Suppose that *check_clique* returns *true*. If a segment $(p, \ldots, p)$ is generated from p by applications of T, then the segment preserves the well-founded property for p.

Proof:

To prove this claim, we must establish a correspondence between the applications of transformation T which produce a rule sequence and the recursive calls of procedure *check_clique*. We do this by defining a structure called a transformation tree.

A *transformation tree* rooted at $p\,1^{(I,Pr)}$ is a tree whose leaves from left to right correspond to a rule sequence constructed by a sequence of steps of transformation T starting from $p\,1$.

A sequence of steps of transformation T starting from p induces a transformation tree as follows:

1. For the empty sequence of steps, it is the node $p^{(\{p\},Pr)}$, where Pr is the property associated with predicate p.
2. Let T be the transformation tree associated with the first k steps of the sequence. Let step k+1 expand (a leaf) q by a rule cycle $C_q = (q, \ldots qi \ldots, q)$. The transformation tree T' associated with the first k+1 steps of the sequence is obtained from T by making $(q^{(Iq,Prq)}, \ldots, qi^{(Iqi,Prqi)}, \ldots, q^{(Iq,Prq)})$, in that order, the children of (the leaf in T) $q^{(Iq,Prq)}$, where Iqi = Iq $\cup$ $\{qi\}$, and $Prqi$ is obtained using the procedure call *tag_prop* $(C_q, Prq, qi, Prqi)$.

We actually prove a slightly stronger claim:

1. If a segment $(p, \ldots, p)$ is produced from p by applications of T, then the segment preserves the well-founded property (say $prop_p$) for p, and
2. Consider the transformation tree for these applications. For each node $q^{(Iq,Prq)}$, there is a call *verify* (q, Iq, Prq) in the execution of the goal *verify* $(p, \{p\}, prop_p)$.

The proof is by induction.

Inductive Hypothesis: Let the claim hold for all segments $(p, \ldots, p)$ produced from a simple cycle by less than k+1 applications of T.

Basis: Consider the simple cycle C = $(p, \ldots, p)$ used initially. This corresponds to the call *verify* $(p, \{p\}, prop_p)$. The call check(C_i, prop) 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* $(C, prop_p, q, prop_q)$ in Step 1 of *verify* which associates the tag $prop_q$, for each node q on this cycle. Also, the tag for node q in the transformation tree is $(\{p\}, prop_q)$. There is a call *verify* $(q, \{p\}, prop_q)$ is Step 2 of *verify*, proving part (2) of the claim for the basis case.

Inductive 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 $(p, \ldots, p)$ 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 correctly tagged (with ignore sets and properties) as in the hypothesis.

Let a node q (tagged with $(Iq, prop_q)$) in the segment be replaced by a simple cycle C1 = $(q, \ldots, q)$. There is a call *verify* $(q, Iq, prop_q)$, according to the hypothesis. This generates a call *check* (C2, $prop_q$) for every simple cycle C2 of the form $(q, \ldots, q)$ which does not contain any node in Iq. Since C1, by definition of transformation T, cannot contain nodes in Iq, there is a call *check* (C1, $prop_q$), and thus, the segment C1 preserves $(q, prop_q)$. From the specification of *tag_prop*, 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 node $q\,1$ in C1, the call to *tag_prop* in Step 1 of the call *verify* $(q, Iq, prop_q)$ associates a property $prop_{q1}$, and the corresponding node in the transformation tree has the tag $(Iq \cup q, prop_{q1})$. The call *verify* $(q, Iq, prop_q)$ in Step 2 generates the call *verify* $(q\,1, Iq \cup q, prop_{q1})$, proving part (2) of the claim.

This completes our proof of Lemma 5.5. []

The following result summarizes our approach.

Corollary 5.6:

Consider a clique C in a database DB = (EDB,IDB,IC). If *check_clique* returns *true*, then every predicate in the clique is strongly safe.

Proof:

This follows immediately from Theorems 5.1 and 5.2. []

We now consider some limitations of Theorem 5.2 and show how it can be strengthened to overcome them.

Example 5.2:

Theorem 5.2 sometimes fails to determine that certain well-founded properties hold, and we present two illustrative programs:

1. p(X,Y) :- q1(X,Y), q2(X,Y)
2. q1(X,Y) :- b(X,Y)
3. q2(X,Y) :- f(X,U), 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, *check_clique* considers whether this property is preserved by the rule cycle (1,3) corresponding to the argument mapping $(p, q2, p)$. Since this cycle considered alone does not preserve boundedness of the first argument of p, using Theorem 5.2 we cannot show that the first argument of p is bounded.

The following example is similar:

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(X,U), 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 $(p, q1, p)$, we see that the first argument of p is monotonically decreasing. Theorem 5.2 cannot show this because it considers the argument mapping $(p, q2, p)$ and the corresponding simple rule cycle (1,3) which does not preserve this property. []

The previous example brought out a limitation of Theorem 5.2. The intuition behind this can be explained as follows. The theorem can establish that no derivation tree constructed using the rules in the clique contains an infinite path, which 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, a weaker condition is sufficient to establish safety: in every derivation tree, there is a set of leaves such that they collectively contain all values which appear in the fact labeling the root, and such that the paths from these leaves to the root are finite. (Recall that we establish strong safety by establishing the safety of every predicate. That is, we consider all derivation trees.) We now use this intuition to present a stronger form of Theorem 5.2. This stronger version can easily be used to refine algorithm *check_clique*, as we later explain.

Let us define the head of a rule to be *covered* by the body if every variable in the head also appears in the body.

Theorem 5.6: Let P be a program containing a rule r, which is of the form: $p() :- q1(), q2(), \ldots, qn()$. Therefore, there is an argument mapping (p, qi, r) generated from each literal occurrence qi in the body. Let *prop* be a well-founded property associated with p.

IF there is some subset of body literals which covers the head, and further is such that for every literal $qi()$ in this subset, every simple cycle $(p, qi, r), \ldots, (\_, p, \_)$ satisfies property *prop*,

THEN we may ignore all simple cycles $(p, qj, r), \ldots, (\_, p, \_)$, qj not in the subset, in verifying whether *prop* is preserved.

Proof: Replace rule r by a rule in which the only body predicates are those in the subset mentioned in the hypothesis. Let us denote the program after this change as P'. Since every rule head is still covered, assumption 1) about programs is not violated by P'. By the correctness of *check_clique*, P' is strongly safe if *check_clique* returns *true* after ignoring the indicated simple cycles (since these cycles do not appear in P'). Thus, every predicate in P' is safe.

Every predicate in P also appears in P'. Further every rule of P is either a rule of P', or a rule of P' with some additional body literals. The additional literals can only restrict the set of computed tuples further. Since every predicate in P' is safe, it follows that every predicate in P is also safe. That is, P is strongly safe. []

Thus, we can make the following modification to the *verify* procedure. As we test simple cycles, if we find a subset satisfying Theorem 5.6, we can ignore the indicated simple cycles. (This is similar to the way we ignore simple cycles which contain a node in the set IGNORE.) On the other hand, we may encounter one of the other simple cycles (of the form $(p, qj, r), \ldots, (\_, p, \_)$, for which the test fails) before we have tested all simple cycles of the form $(p, qi, r), \ldots, (\_, p, \_)$, where qi is in the covering subset (assuming that some such 'good' subset exists). So if *verify* fails for a simple cycle, rather than aborting, this should be recorded and *verify* should continue testing other simple cycles till all candidate qi's (from the corresponding rule) 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* for a simple cycle allows us to avoid repeating the test if it is required again subsequently.)

We remark that Theorem 5.6 is not only more powerful than Theorem 5.2, it is often considerably more efficient. For example, consider the case when the body literals 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.

We would like to emphasize the extensible nature of this approach. As we noted earlier, two programs, *check* and *tag_prop* are assumed to be given as inputs to *check_clique*, and using more sophisticated versions of these programs that can deal with a richer class of well-founded orderings clearly increases the scope of *check_clique*. Also, we have not addressed the issue of how to infer monotonicity constraints. Some work in this direction has been presented in [Ullman and Van Gelder 85], and we can take advantage of those (and other, similar) results to infer constraints. A more subtle point is that we need to identify bounded arguments to check whether orderings are well-founded. There are some obvious ways of doing this. For example, the literal X < 10 in a rule tells us that X is bounded below, and b(X), where b is a finite base relation, tells us that X is bounded (equivalently, safe). Recall that the methods presented in this paper focus on *strong safety*. Other methods for inferring safety, for example, those presented in [Ramakrishnan et al. 87] for inferring safety using only FCs, can be used in our analysis to identify arguments that are bounded.

In concluding this section, we observe a weakness of the framework which represents 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,X)_? where p is defined as follows:

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 [Beeri and Ramakrishnan 87], they cannot propagate equalities between unbound 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)_? appeared by adding a literal *finite*(X) to the rule containing the literal p(X,X)_? 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)_? 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 a literal *finite*().)

6. Related Work

Some results relating to these problems have been presented in the context of testing the applicability *top-down capture rules* [Sagiv and Ullman 84, Ullman and Van Gelder 85, Afrati et al. 86]. These problems were also addressed in [Ramakrishnan et al. 87]. In this paper, we make use of, and extend, this earlier work.

This work can be seen as providing a framework for testing safety and effective computability of recursive programs, in some ways analoguous to the *capture rules* framework of Ullman [Ullman 85]. The capture rules framework considers the program represented as a *rule/goal graph*, and allows the design of *capture rules* which 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 which 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.

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 (analoguous 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. [Ullman and Van Gelder 85]. In our framework, these procedures include algorithm *check_clique*, the two auxiliary procedures called *check_clique*, and other procedures for inferring monotonicity constraints and *boundedness* of predicates.

We now examine the relative merits of the two approaches briefly. Different capture rules can be expressed in terms of a set of sideways information passing strategies (sips) which are implemented by rewriting the program [Beeri and Ramakrishnan 87]. The applicability of a capture rule must be shown by presenting a *substantiation*, that is, a program which computes the (sub)query according to the capture rule. The bottom-up evaluation of the rewritten program provides a potential *substantiation*. The correctness of the rewriting transformations implies that the set of answers is correctly computed by the bottom-up evaluation, but we must still establish safety of all intermediate results and termination. This is the focus of the present paper. Thus, in our framework, the range of sideways information passing strategies (sips) describes the set of possible capture rules, and there is a single procedure (based on *check_clique*) which tests for the applicablility of all capture rules by testing the rewritten program for safety and effective computability. Further, the substantiation is always a program which is evaluated bottom-up. In contrast, in the capture rules framework, each new capture rule must be explicitly represented along with its substantiation. Testing applicability is specific to a given capture rule, and the substantiation need not always be a program which is evaluated bottom-up.

As we remarked earlier, the results here can be used to show the "applicability" of capture rules in the literature, and can also be used to show the safety and effective computability of some programs which cannot be "captured" using known capture rules.

We now consider the testing of programs produced by rewriting algorithms like the Magic Sets algorithm. The Magic Sets strategy [Bancilhon et al. 86] 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 [Beeri and Ramakrishnan 87].

Conceptually, we first rewrite the program and then test it for safety and effective computability. A potential problem is that the rewritten program typically has much larger cliques than the original program since the rewriting introduces rules which often combine cliques in the original program. Thus, even if 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. However, we can take advantage of the structure of the transformation to efficiently test rewritten programs for safety and effective computability. The *adorned program* is an intermediate program produced by the Magic sets algorithm, and has a clique structure which is similar to that

of the given program. We show how the rewritten program can be analyzed by considering the cliques in the adorned program, and without directly considering the cliques in the rewritten program [Krishnamurthy et al. 87]. We have not presented these results here due to space constraints, but they suggest that the framework here is robust and can provide the basis for a practical system.

In this paper, we do not consider the problem of how to establish that a node is bounded. That is, our algorithms test for safety in conjunction with effective computability, but they cannot be used to determine if a query is simply safe (but, possibly, not effectively computable). Our results clearly provide sufficient conditions for safety, but it is possible to devise weaker conditions if only safety is of interest. (We note that some of the results in the literature for detecting safety, for example [Ramakrishnan et al. 87, Zaniolo 85, Van Gelder and Topor 87], could be used for this purpose.)

7. Conclusions

The important contribution of this paper is an algorithm (*check_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 which are assumed to be available, such as *check*, *tag_prop*, and auxiliary procedures for inferring boundedness and monotonicity constraints. The procedures *check* and *tag_prop* represent abstractions of operations that are implicit in any safety analysis involving monotonicity constraints, and in this paper they are isolated and 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.

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. A first step in these directions is taken in [Ullman and Van Gelder 85]. 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.

8. References

[Afrati et al. 86]
"Convergence of Sideways Query Evaluation", F. Afrati, C. Papadimitriou, G. Papageorgiou, A. Roussou, Y. Sagiv and J.D. Ullman, *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986, pp 24-30.*

[Bancilhon et al. 86]
"Magic Sets and Other Strange Ways to Implement Logic Programs", F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986, pp 1-16.*

[Bancilhon and Ramakrishnan 86]
"An Amateur's Introduction to Recursive Query Processing Strategies", F. Bancilhon and R. Ramakrishnan, *Proc. SIGMOD 86, pp 16-52.*

[Bancilhon and Ramakrishnan 87]
"Performance Evaluation of Data Intensive Logic Programs", F. Bancilhon and R. Ramakrishnan, *To appear in Foundations of Deductive Databases and Logic Programming, Ed. J. Minker, Morgan Kaufman.*

[Gaifman 86]
Personal communication, reported by Y. Sagiv.

[Krishnamurthy et al. 87]
"Testing Safety and Effective Computability of Magic Programs", R. Krishnamurthy, R. Ramakrishnan and O. Shmueli, *Manuscript.*

[Lloyd 84]
"Foundations of Logic Programming," J. W. Lloyd, *Springer-Verlag, 1984.*

[Ramakrishnan et al. 87]
"Safety of Horn Clauses with Infinite Relations", R. Ramakrishnan, F. Bancilhon and A. Silberschatz, *Proc. 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1987, pp 328-339.*

[Sagiv and Ullman 84]
"Complexity of a Top-Town Capture Rule" Y. Sagiv and J.D. Ullman, *STAN-CS-84-1009, Department of Computer Science, Stanford University, 1984.*

[Shmueli 87]
"Decidability and Expressiveness Aspects of Logic Queries", O. Shmueli, *Proc. 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1987, pp 237-249.*

[Ullman 85]
"Implementation of Logical Query Languages for Databases," J. Ullman, *TODS, Vol. 10, No. 3, pp. 289-321, 1985.*

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

[Zaniolo 86]
"Safety and Compilation of Non-Recursive Horn Clauses", C. Zaniolo, *Proc. First Intl. Conference on Expert Database Systems, Charleston, 1986.*