Safety of Recursive Horn Clauses With Infinite Relations

(Extended Abstract)

Raghu Ramakrishnan (1,2)  
Francios Bancilhon (2)  
Avi Silberschatz (1)

ABSTRACT

A database query is said to be safe if its result consists of a finite set of tuples. If a query is expressed using a set of pure Horn Clauses, the problem of determining whether it is safe is in general undecidable. In this paper, we show that the problem is decidable when terms involving function symbols (including arithmetic) are represented as distinct occurrences of uninterpreted infinite predicates over which certain finiteness dependencies hold. We present a sufficient condition for safety when some monotonicity constraints also hold.

1 Introduction

A Horn Clause is a rule of the form \( p(t) \rightarrow q_1(t_1), q_2(t_2), \ldots, q_n(t_n) \), where \( p(t) \) and the \( q_i(t_i) \)'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 an m-ary function symbol followed by m 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 \( q_i(t_i) \)'s are true, i.e., the tuple \( t \) is in the relation \( p \) if for all \( t \) tuple \( t_i \) is in relation \( q_i \).

We partition the given set of Horn Clauses into a set of facts which are stored in the base predicates (the Extensional Database or EDB), and a set of 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. A set of integrity constraints (IC) may 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).

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. A ground term is a term containing no variables. The EDB may contain predicates which have an infinite number of facts. These infinite predicates are used to represent arithmetic operations and terms generated by function symbols.

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, \( p \) refers to the \( i \)th attribute of predicate \( p \). Variables are denoted by uppercase letters and constants are denoted by numerals.

A detailed description of Horn Clauses and formal definitions can be found in [Bancilhon and Ramakrishnan 86a].

Example 1

Consider a database which contains information about parent-child relationships. The information is kept in a single base relation called parent. We define a relation ancestor over the relation parent using two rules.
parent(ceos,adam)
parent(abel,adam)
parent(ceos,eve)
parent(abel,eve)
parent(sem,abel)
ancestor(X,Y,J) - ancestor(X,Z,J),
parent(Z,Y), successor(I,J)
ancestor(X,Y,I) - parent(X,Y)

The meaning of ancestor(X,Y,J) is that Y is an J'th level ancestor of X. Thus, abel is a 1st level ancestor (parent) of sem, and adam is a 2nd level ancestor (grandparent) of sem. The relation successor(I,J) is an infinite relation defined by J = I+1, for all positive integers I. Thus, the EDB contains the finite base predicate parent and the infinite base predicate successor, and the IDB contains the predicate ancestor. 

A query is said to be 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 dependencies and monotonicity constraints. We begin by considering only finiteness dependencies and extend the analysis to cover the use of monotonicity constraints in Section 6.

A finiteness dependency (FD) over a base predicate r is of the form X → Y, where X and Y are sets of attributes. An instance of r satisfies this dependency iff the following property holds: if r(X) is finite, then r(Y) is finite. 

Note that this definition is strictly weaker than the traditional definition of a functional dependency, it holds trivially for all finite predicates.

Example 2
Consider the predicate f(X,Y). If we define f(X,Y) by Y = 2*X, then we have the finiteness dependencies f_2 → f_1 and f_1 → f_2. If we define f(X,Y) by X < 0 and Y > 0, there is no finiteness dependency between the attributes of "f". If we define f(X,Y) by X > 0 and Y = 0 or 5, we have the finiteness dependency f_1 → f_2.

The Armstrong axioms for functional dependencies also characterize finiteness dependencies.

Theorem 1: Let r be a predicate, and X, Y and Z be sets of attributes of r. The following set of axioms is sound and complete for finiteness dependencies.

1. Reflexivity: If X contains Y, X → Y holds.
2. Augmentation: If X → Y holds, then XZ → YZ holds.
3. Transitivity: If X → Y and Y → Z hold, then X → Z holds.

Example 3
Consider a database consisting of a finite base predicate b and an infinite base predicate t, and the following rules:
r(X) - t(X,Y), r(Y)
r(X) - b(X)

Consider the query r(X)? We can apply the first rule (after applying the second infinitely), possibly generating a number of new tuples for r. Thus, the relation computed for r is potentially infinite. The given query is unsafe.

Example 4
Consider a modified version of Example 3 where a new finite base predicate a is included and the rules have been modified as follows:
r(X) - t(X,Y), r(Y), a(Y)
r(X) - b(X)

Let the FD f_2 → f_1 hold.

The base predicate a in the first rule ensures that no matter how often it is applied, the set of possible values for Y is finite. Since Y finitely determines X (because of the dependency f_2 → f_1), it also ensures that the set of possible values for X is finite. Thus, the query is safe.

Note that this query becomes unsafe if we delete the term a(Y) from the first rule. While the finiteness dependency ensures that we generate only a finite number of new values at each step, we could potentially apply the rule infinitely often to produce an infinite number of possible values for X.

The above example illustrates that if an argument place in the query can receive values from an infinite predicate, it must also be finitely determined by some collection of finite database predicates if it is to be safe. This must hold no matter which subset of rules are used to generate the bindings. For instance, if r were defined by all the rules in the previous two examples, the rules in the first example would make r unsafe despite the fact that the rules in the second example are, in themselves, safe. These ideas are developed rigorously in subsequent sections.

Due to space limitations, no proofs are presented in this abstract. The proofs of the lemmas and theorems presented here may be found in [Ramakrishnan et al. 87]
2. A Canonical Form for Horn Clauses

We present an algorithm which takes a set of Horn Clauses and produces a set of Horn Clauses in a canonical form in which all arguments are variables and all occurrences of a function symbol (in the original set of clauses) are replaced by a unique occurrence of an infinite relation.

Example 5

An example of a rule in canonical form is

\[ r(X) \rightarrow g(X,Y), b(Y,Z) \]

while the following rule is not in canonical form because the constant '5' and the term \([X,Y]\) appear in it

\[ r(Z) \rightarrow g([X,Y],5) \]

The algorithm below transforms a given set of Horn Clauses into canonical form. In doing so, much information about the original program is discarded. This is deliberate, we discard all information which is not utilised in the subsequent safety analysis. We later state a necessary and sufficient condition for the transformed set of clauses to be safe. Clearly, this is only a sufficient condition for the original set of clauses.

Algorithm 1

Repeat the following steps until all clauses are canonical

1. If a rule contains a constant, do the following
   Replace the constant by a new variable, say \( X \). Add the literal \( b(X) \) to the body of the rule, where \( b \) is the name of a new (finite) EDB predicate. If the same constant appears in several places, use the same predicate.

2. If a rule contains a term with a function symbol, do the following
   Replace the term by a new variable, say \( X \). Let the term be \( g(Y_1, \ldots, Y_k) \). Add the literal \( h(Y_1, \ldots, Y_k, X) \) to the body of the rule, where \( h \) is the name of a new (infinite) EDB predicate.

Example 6

Consider the following query

\[ r(X,Y) \rightarrow p(X,5), r(5,Y) \]

\[ r(X,Y) \rightarrow a(X,Y) \]

\[ r(X,2) \]

Algorithm 1 would produce the following set of rules

\[ r(X,Y) \rightarrow p(X,U), r(V,Y), b(U), b(V) \]

\[ r(X,Y) \rightarrow a(X,Y) \]

\[ q(X) \rightarrow r(X,U), c(U) \]

\[ q(X) \]

In the above rules, \( b \) and \( c \) are base predicates. In particular, the following facts correspond to the given set of clauses.

\[ b(5) \]

\[ c(2) \]

We have introduced additional database predicates \( b \) and \( c \), each containing a single tuple. Neither the IDB nor the query contains any constants.

Example 7

The following is a program that concatenates two lists

\[ concat([X,Y],Z,[X,U]) \rightarrow concat(Y,Z,U) \]

\[ concat([],Z,Z) \]

We rewrite it as follows

\[ concat(V,Z,W) \rightarrow concat(Y,Z,U), \]

\[ h_1(X,Y,V), h_2(X,U,W) \]

\[ concat(N,Z,Z) \rightarrow nil(M) \]

\[ nil([]) \]

The predicates \( h_1 \) and \( h_2 \) are infinite base predicates corresponding to the function symbol \( 1 \) Conceptually, they contain all triples of lists such that the third is the concatenation of the first two. The predicate \( nil \) contains the single element denoting the empty list. The empty list is a constant, and thus we use the same device that we described earlier for handling constants.

Theorem 2: Let \( q \) be a query defined using a set of Horn Clauses \( H \), and let \( H' \) be the canonical set of clauses corresponding to \( H \) (with \( q' \) corresponding to \( q \)) obtained using Algorithm 1. Then, \( q \) is safe if \( q' \) is safe.

We note that the converse of Theorem 2 does not hold, as the following example demonstrates.

Example 8

Consider the following set of clauses

\[ r(X) \rightarrow p(Y), q(Y), integer(X) \]

\[ p([11111]) \rightarrow q([11111]) \rightarrow r(X) \]

where \( integer(X) \) denotes that \( X \) is an integer ('integer' is an infinite base predicate containing all integers). Predicates \( p \) and \( q \) both contain a single tuple. The tuple in \( p \) is a list of length 1, and the tuple in \( q \) is a list of length 2. Thus, the first rule never succeeds and the relation for predicate \( r \) is empty. The query \( r(X) \) is therefore safe.

Algorithm 1 translates this set of clauses to the following set.
where $h_1$, $h_2$ and $h_3$ are infinite base predicates corresponding to $I$. The query $r(X)$ is unsafe, no matter what FDs we consider to hold over $h$. To see this, let $h_1$, $h_2$ and $h_3$ contain the tuples $(1, [1, [1]]$ and $(1, [1, [1]]$. This implies that both $p$ and $q$ now contain $'1'$, and so $r(X)$ succeeds for all integers $X$. While this is not consistent with the semantics of $I$, it satisfies any FDs we specify, since we have chosen finite instances for $h_1$, $h_2$ and $h_3$.

### Safety Analysis Using Finiteness Dependencies

Given a set $H$ of canonical Horn clauses and a query $q$, we test $q$ for safety as follows: From each clause in $H$ we obtain one or more (Horn) clauses in propositional logic for each argument place. Let the set of all such derived clauses be called $\text{And-Or } H$. Intuitively, each clause in $\text{And-Or } H$ describes one way in which a given argument place could receive a potentially infinite set of bindings (become unsafe) through some rule in the set of Horn Clauses $H$. We analyse $\text{And-Or } H$ and show that $H$ is safe if $\text{And-Or } H$ satisfies certain conditions.

We begin by presenting an algorithm for deriving $\text{And-Or } H$ from $H$. This is done in two steps by deriving a set of Horn Clauses $H^*$ from $H$ and then deriving $\text{And-Or } H$ from $H^*$.

The set $H^*$ consists of adorned rules, which are defined below. An adornment of an $m$-place literal is a string of $m$ bs and $fs$ such that if a variable $X$ occurs in argument places $i$ and $j$, the $i$th and $j$th elements in the string are identical. An adorned literal is a literal with the predicate name superscripted by an adornment. Consider a rule $p \leftarrow q_1, \ldots, q_m$ in $H$, where the head literal $p$ has $n$ arguments. An adorned version of this rule is one in which the head literal is adorned. An adorned version of a rule represents the rule when it is used to compute values for the free arguments, given constants corresponding to the bound arguments. Since there are at most $2^n$ adornments for the $n$-place head predicate, there are at most $2^n$ adorned versions of a given rule.

We now rename variables so that each variable name appears in exactly one rule. We also rename all predicates in rule bodies (except finite base predicates) to make the names unique. The name of a literal is taken to be the corresponding predicate name, in italics.

### Example 9

Consider the following set of canonical Horn Clauses $H$:

\[
\begin{align*}
r(X,Y) & \quad f(X,U,V), r(U,V), b(U,Y) \\
r(X,Y) & \quad b(X,Y)
\end{align*}
\]

The corresponding set $H^*$ of adorned rules consists of:

\[
\begin{align*}
r^f(X1,Y1) & \quad f f(X1,U1,V1), r 1(U1,V1), b(U1,Y1) \\
r^f(X2,Y2) & \quad - b(X2,Y2) \\
r^f(X3,Y3) & \quad f f(X3,U3,V3), r 2(U3,V3), b(U3,Y3) \\
r^f(X4,Y4) & \quad - b(X4,Y4) \\
r^f(X5,Y5) & \quad f f(X5,U5,V5), r 3(U5,V5), b(U5,Y5) \\
r^f(X6,Y6) & \quad - b(X6,Y6) \\
r^f(X7,Y7) & \quad f f(X7,U7,V7), r 4(U7,V7), b(U7,Y7) \\
r^f(X8,Y8) & \quad - b(X8,Y8)
\end{align*}
\]

The algorithm for deriving $H^*$ from $H$ is simple and will not be presented here.

We now consider the second step in the derivation of $\text{And-Or } H$ from the canonical set of clauses $H$.

The set $\text{And-Or } H$ consists of rules of the form $p \rightarrow q_1, \ldots, q_m$, where $p$ and all the $q_i$s are propositional variables. Each of these variables may thus take on the value '0' or '1'. 'Unsafe' is equivalent to '1'. Each propositional variable in $\text{And-Or } H$ corresponds to an argument or a variable in $H$ (or an order of evaluation by which values can be produced for an argument or variable, as will be seen later). Intuitively, the above rule may be read as follows: $p$ is unsafe if all the $q_i$s are unsafe. Each rule in $\text{And-Or } H$ with head $p$ represents one way in which $p$ can receive values in a computation of $H$. $p$ can receive a value through this rule if its value is contained in each of the $q_i$s. Thus, $p$ can receive only a finite number of values through this rule if at least one of the $q_i$s can only receive a finite number of values, otherwise, $p$ can receive an infinite number of values ($p$ is unsafe). $p$ is safe if none of the rules for $p$ make it unsafe.

If the propositional literal corresponding to some argument position or variable evaluates to '1' in the least fixpoint solution of $\text{And-Or } H$, then that argument position or variable is unsafe (The converse is not true - something which evaluates to '0' is not necessarily safe).

We present below an algorithm for deriving the set $\text{And-Or } H$ from the adorned set of clauses $H^*$. We illustrate it with an example and then discuss it informally.

### Algorithm 2

**Input:** A set of adorned Horn clauses $H^*$ and a set of FDs over the EDB predicates

**Output:** $\text{And-Or } H$
For each adorned rule \( p^e(t) - q_i(t_1), \ldots, q_n(t_n) \) in \( H^* \) do

begin

1. For each position \( k \) in 'a' (in the head \( p^a \)) do
   - If the \( k \)th position in adornment 'a' is \( b \),
     Add 'pfk - 0' to And-Or \( H \)
   - If the \( k \)th position in 'a' is \( f \), and the \( k \)th argument of \( p^a \) is \( X \),
     Add 'pfk - X' to And-Or \( H \)

2. For each variable \( X \) that appears in the adorned rule, let \( CX \) be the conjunct of all \( q_i \) where \( X \) appears in the \( k \)th argument of body literal \( q(t) \)
   - If at least one of the \( q_i \)s in the conjunct is a finite base literal,
     Add 'X - 0' to And-Or \( H \)
   - Else,
     Add 'X - CX' to And-Or \( H \)

3. For each derived literal \( q \) with \( m \) arguments in the body
   - For each argument position \( k \) do
     Let \( L \) denote the conjunct of all possible \( qf^1 \) where 'a1' is an adornment of \( q \) with the \( k \)th element being \( f \). Thus, the conjunct \( L \) has at most \( 2^{m-1} \) elements (it may have less if \( q \) has the same variable in two arguments)
     Add 'qak - L' to And-Or \( H \)
     For each of the elements \( qf^1 \) in the conjunct \( L \), for each variable \( Y \) that appears in an argument of \( q \) corresponding to a \( b \) in the adornment 'a1',
     Add 'qf^1 - Y' to And-Or \( H \)
     Further, let \( q = l_f \) (predicate name \( l \) with suffix \( f \)). For each \( qf^1 \),
     Add 'qf^1 - l^1f' to And-Or \( H \)
     (We observe that \( l^1f \) is the name of a head literal in \( H^* \))

4. For every occurrence \( f \) of some infinite base predicate, \( m \) arguments, in the body
   - For each argument position \( k \) do
     Let there be \( n \) FDs between the arguments of \( f \) that determine the \( k \)th argument, and let the \( i \)th such FD be denoted by \( F_i \rightarrow f_k \)
     Add 'f_k - f_k' to And-Or \( H \)
     (It is possible that no FDs are known to determine \( f_k \). In that case, add 'f_k - 1' to And-Or \( H \))
     For each variable \( Y \) which occurs in some position \( j \) in \( F_i \),
     Add 'f_k - Y' to And-Or \( H \)

end \[ \]

We present an example to illustrate Algorithm 2

Example 10

Consider the following adorned Horn Clause in \( H^* \)
\( r^f(X1,Y1) - f 1(X1,U1,V1), r 1(U1,V1), b(U1,Y1) \)
with the finiteness dependency \( f 2, f 3 \rightarrow f 1 \) given

Algorithm 2 generates the following rules from this adorned clause, and adds them to And-Or \( H \)

Step 1 Since both positions in the head adornment are \( f \), we get
\( r 1f - X1 \)
\( r 1f - Y1 \)

Step 2 We generate rules defining each variable
\( X1 - f 1r \)
\( Y1 - 0 \)
\( U1 - 0 \)
\( V1 - f 13, r 12 \)

Step 3 We generate rules corresponding to argument places of derived predicate occurrences. Consider the occurrence \( r 1f \) of the derived predicate \( r \). Since there are two arguments, in considering the first argument, the possible adornments are \( ff \) and \( fb \)
\( r 1f - r 1fb, r 1ff \)

Consider \( r 1fb \) Variable \( V1 \) appears in the second argument, which is bound. Thus, using step 3 again, we have the rule,
\( r 1fb - V \)

We also have the rule,
\( r 1fb - r 1f \)

Consider \( r 1ff \) Since there are no bound arguments, the only rule is
\( r 1ff - r 1f \)

Proceeding similarly, we obtain the following rules for \( r 12 \),
\( r 12 - r 1ff, r 12f \)
\( r 1ff - U \)
\( r 1ff - r 1f \)
\( r 1ff - r 1f \)

Step 4 We generate rules for the argument positions of infinite base predicate occurrences. Consider the occurrence \( f 1f \) of the infinite base predicate \( f \). The set of known FDs is
\( 1 f 12, f 13 \rightarrow f 11 \)
Thus, using step 4, we get the rule,
\[ f_{11} \rightarrow f_{11} \]
Since variable U1 appears in the second argument of \( f_1 \) and V1 appears in the third argument, and both these arguments appear in the left hand side of the FD, using step 4 again, we get the rules,
\[ f_{11} \rightarrow \text{U1} \]
\[ f_{11} \rightarrow \text{V1} \]
Since there are no FDs defining \( f_{12} \) and \( f_{13} \), we get the rules,
\[ f_{12} \rightarrow 1 \]
\[ f_{13} \rightarrow 1 \]

We now discuss the above algorithm informally. Step 1 introduces rules for head literals \( p^* \). If an argument position is bound, it is safe, if not, it is unsafe if the variable which occurs in that position is unsafe.

Step 2 in Algorithm 2 introduces rules to determine the safety of variables. A variable is unsafe iff all the literal arguments that it appears in are unsafe.

Step 3 in the algorithm deals with literal occurrences in the right hand side of an adorned rule. Consider a literal occurrence \( q \) with \( n \) arguments. Let \( q = l_1 \) and let 'al' be an adornment of \( q \) such that the bound arguments contain safe variables. We can generate all possible values for the free arguments by using rules having \( l^* \) as the head predicate. Thus, \( q_k \) is safe if \( l^k \) is safe. Thus corresponds to the use of a sideways information passing strategy - we solve \( q \) by solving an adornmed version \( q^k \). We denote the conjunct of all possible \( q^k \)'s by \( L \).

Thus, \( q_k \) is unsafe if the conjunct \( L \) is unsafe, i.e., every element in the conjunct is unsafe.

Consider an element \( q^k \) in the conjunct \( L \). It denotes whether or not the \( k \)th argument of \( q \) is safe given that each of the arguments bound in 'al' is safe. It is thus inapplicable if \( Y \) is unsafe for some \( Y \) which appears in a bound argument of \( q \). This is expressed by the rules \( 'q^k \rightarrow Y' \). (There is the possibility of a circular argument here - we could argue the safety of \( X \) by assuming the safety of \( Y \) and then the safety of \( Y \) by assuming the safety of \( X \). We avoid this by using the notion of a forward cycle.) We discuss this point in greater detail when we describe AND-graphs \( \text{And}_H(p) \).)

Even if all variables in bound arguments of 'al' are safe, \( q^k \) is unsafe if \( l^k \) is unsafe. We express this by the rule \( 'q^k \rightarrow l^k' \).

Step 4 translates the information present in the finiteness dependencies. Let \( f \) be the name of a literal occurrence in the body whose predicate name is the name of an infinite base predicate. Further, let \( f_k \) be determined by a set of \( n \) FDs. Let us consider what can be inferred about the safety of \( f_k \) from the set of FDs. If each element in the antecedent of one of the \( n \) FDs is safe, then that FD can be used to infer that \( f_k \) is safe. The set of bindings for \( f_k \) is thus potentially infinite if none of the FDs restricts it to a finite set of values. Let the set of bindings computed by the \( i \)th FD be \( f_{ki} \). We can then say that the set of bindings for \( f_k \) is potentially unsafe if \( f_{ki} \) is unsafe for all FDs \( 'i' \) which determine \( f_k \). This is represented by the rule \( 'f_k \rightarrow f_{k1}, \ldots, f_{kn}' \).

Further, we know that \( f_{ki} \) is unsafe if even one of the antecedents of the \( i \)th FD is unsafe, since the FD then becomes inapplicable. Thus, we have the rules \( 'f_{ki} \rightarrow Y' \), where \( Y \) is the variable in the \( i \)th argument of \( f \) and the \( i \)th FD is of the form \( 'f_{ki}, \ldots, f_{kn}' \). These rules summarize the information present in the FDs.

We now establish several results concerning \( \text{And} - \text{Or}_H \) and demonstrate that \( H \) is safe if \( \text{And} - \text{Or}_H \) satisfies certain constraints.

The set \( \text{And} - \text{Or}_H \) may be viewed as an AND-OR graph with each literal as an OR node and each rule as an AND node. The children of an OR node \( q \) are the rule nodes for each rule in \( \text{And} - \text{Or}_H \) with head \( q \). The children of an AND node are the OR nodes corresponding to each literal on the right hand side. Given an AND-OR graph with root \( p \), it is straightforward to decompose it into a collection of AND graphs rooted at \( p \).

The set \( \text{And} - \text{Or}_H \) contains a node \( p \) corresponding to every variable and literal argument in \( H^* \). Let us associate a binding set with each node in \( \text{And} - \text{Or}_H \). This denotes the set of values (constants) to which the corresponding variable or argument may be bound by using the rules in \( H^* \).

Consider a computation of the answer for a query, using the rules in \( H \). We view a computation as a sequence of rule applications. If some of the head arguments are bound, we use an adorned version of the rule in which these arguments are bound. We then obtain bindings for the arguments of body literals by solving them. In solving a body literal, some arguments may be bound by the previously solved literals in this rule. Further, if some arguments are bound, we may use FDs to bind more arguments. We only need to solve an adorned version of this literal in which all \( b \)s correspond to bound arguments. After obtaining some bindings for all arguments of body literals, we can use the rule to produce bindings for the free arguments in the head. Note that this view is sufficiently general to describe both top-down and bottom-up computation strategies.

Initially, only the binding sets associated with base predicates, the bound arguments in the query, and the variables appearing in them are non-empty. A value can be added to a binding set only by the application of a rule in \( H^* \). Each rule in \( \text{And} - \text{Or}_H \) with head \( p \) corresponds to a way in which a binding can be computed for \( p \).
Given a variable or argument position $p$, any computation of a binding for it (a value in its binding set) can thus be represented as an AND-graph rooted at $p$, in which the nodes are propositional variables appearing in $\text{And-Or}_H$. A binding path for $p$ is a sequence of propositional variables (in $\text{And-Or}_H$) such that $p$ is the first element and $q$ is the successor of $q'$ only if values are computed for the binding set of $q$ using values in the binding set of $q'$. Note that a binding path might be cyclic (some node appears twice in the path). Let us define a path $A$ to be contained in an AND-graph $G$ if they have the same root and $A$ can be obtained from graph $G$ by deleting some nodes and edges. We call an AND-graph constructed using the rules in $\text{And-Or}_H$ a cover of a binding path $B$ if it contains $B$.

**Lemma 1:**
1. Every binding path $B$ for $p$ is contained in one or more AND-graphs with root $p$ constructed using the rules in $\text{And-Or}_H$.
2. Every AND-graph with root $p$ constructed using the rules in $\text{And-Or}_H$ contains a binding path for $p$.

**Lemma 2:** $p$ has a finite binding set ($p$ is safe) if for every binding path $B$ of $p$, every cover of $B$ contains a binding path which produces a finite set of bindings.

**Lemma 3:** $p$ is safe if every AND-graph with root $p$ constructed from the rules in $\text{And-Or}_H$ contains a binding path which produces a finite set of bindings.

To analyze (the potentially infinite) AND-graphs that can be constructed from the rules in $\text{And-Or}_H$, we consider a (finite) collection of finite AND-graphs rooted at $p$. We call such a graph $\text{And}_H(p)$. (There could be more than one graph $\text{And}_H(p)$.)

Each AND graph $\text{And}_H(p)$ is essentially a subset of the set of rules in $\text{And-Or}_H$. We construct a graph $\text{And}_H(p)$ from $\text{And-Or}_H$ by selecting a rule with $p$, as the head and adding exactly one rule for each new element (other than 0 or 1) that appears on the right hand side of this rule, and recursively, on the right hand side of each of these new rules. Thus each rule of a graph $\text{And}_H(p)$ has a unique head predicate.

Let us define an AND-graph $A$ to be an initial subgraph of an AND-graph $B$ if there is a bijection from $A$ to a subset of $B$ such that the root of $A$ is mapped to the root of $B$, and if $p$ is mapped to $q$, the children of $p$ are mapped to the children of $q$.

**Lemma 4:**
1. Every graph $\text{And}_H(p)$ is an initial subgraph of some AND-graph with root $p$ constructed from the rules in $\text{And-Or}_H$.
2. Every AND-graph with root $p$ constructed from the rules in $\text{And-Or}_H$ has some graph $\text{And}_H(p)$ as an initial subgraph.

**Lemma 5:**
1. If a graph $\text{And}_H(p)$ contains a binding path that produces a finite set of bindings, then every AND-graph of which it is an initial subgraph contains this path.
2. If a graph $\text{And}_H(p)$ does not contain a binding path that produces a finite set of bindings, then there is an AND-graph of which it is an initial subgraph which does not contain such a binding path.

Consider a node $f_k$, where $f$ corresponds to an infinite base relation. We use the convention of calling such nodes $f$-nodes, for brevity. We call an edge $p_k - X$ a forward edge if $p$ is a head literal in $H$. A cycle containing a forward edge is called a forward cycle.

**Lemma 6:** Any binding which is computed by a graph $\text{And}_H(p)$ which contains a forward cycle that has no $f$-nodes on it is also computed by another graph $\text{And}_H(p)$ which does not contain such a cycle.

We now introduce several important definitions. We begin with the notion of subgraph replacement at a node $r$ in a graph $\text{And}_H(p)$. Given a graph $\text{And}_H(p)$, we can obtain another graph $\text{And}_H(p)$ as follows: Construct the given graph $\text{And}_H(p)$ by starting with the rule for $p$ and proceeding as before. However, when the term $r$ is encountered, use one of the other rules for $r$, and proceed to complete the construction. This transformation of a given graph $\text{And}_H(p)$ into another graph $\text{And}_H(p)$ which differs from the first only in the subgraph rooted at $r$ is called subgraph replacement at node $r$.

We call a graph $\text{And}_H(p)$ unsafe if applications of this graph can produce an infinite set of bindings for $p$ (i.e., it does not contain a binding path which produces a finite binding set). An unsafe $f$-node is an $f$-node $f_k$ (in the notation of Algorithm 2) which is unsafe. The point to note here is that such a node could be defined by more than one rule, all of which were derived from the same clause in the original set of Horn Clauses $H$. Thus they all refer to the application of a single rule in $H$.

However, each of them appears in a different graph $\text{And}_H(p)$! So we need to be careful when reasoning about their safety in an application of a graph $\text{And}_H(p)$. Consider an $f$-node in a given graph $\text{And}_H(p)$. It is unsafe if its (single) child is unsafe in the given graph $\text{And}_H(p)$ or in any of the graphs obtained from it by subgraph replacement at the child node.
unsafe l-node is defined to be a node q^l whose (single)
child is unsafe in the given graph And^H(p) or in any of
the graphs obtained from it by subgraph replacement at
the child node.

Finally, a safety ensuring 0-node in a given graph
And^H(p) is a 0-node (a node 'O') such that there is
some path from it to p which is free of unsafe l-nodes
and unsafe l-nodes. It is easy to see that the presence of
such a node ensures that the graph is safe since the set
of bindings for p is limited to values which are in a
finite base relation or can be derived from these values
by applying finiteness dependencies a finite number of
times. The path from such a node to p is a binding path
which produces only a finite number of bindings.

We establish the following theorem:

Theorem 3: p is safe if every possible subset
And^H(p) satisfies the following subset condition
And^H(p), viewed as a graph, must contain either a 0-
ode or a forward cycle containing no l-nodes.

Now let us consider why it is not necessary that each
subset And^H(p) satisfy the subset condition. Intu-
tively, unsafeness arises when a head variable derives
its binding through an FD in an infinite relation f This
could lead to an infinite set of bindings for the head
variable. However, a potentially dangerous binding path
could turn out to be harmless if one of the variables in
it has no possible bindings. This could happen if it occurs
in a predicate which has no rules defining it or in a
recursive predicate that is not grounded.

Example 11

The following rule in a set of rules H would cause H to
fail the subset condition:

r(X) - f(X,Y), r(Y)
FD f_2 \rightarrow f_1

However, H is safe if it contains no other rule with head
r^l.

We now describe a modification to the derivation of
And-Or^H which makes the subset condition on the
sets And^H(p) both necessary and sufficient for the
safety of H.

Algorithm 3

begin
Compute a set of propositional Horn clauses T from H
as follows:

For each base predicate p (finite or infinite) add the rule
'p \rightarrow 1' to T.
For each rule p(t) - q_1(t_1), \ldots, q_n(t_n) in H, add 'p
\rightarrow q_1, \ldots, q_n' to T.
Compute the LFP of T. Let T_0 be the set of all predic-
ates assigned the value 0.
Modify And-Or^H by deleting all rules with head p,
where p is in T_0.
end.

Lemma 7: A predicate is in T_0 iff the relation computed
for it using the rules in H is empty for all EDBs.

Theorem 4: p is safe iff every possible subset
And^H(p), computed using the modified set And-Or^H,
satisfies the subset condition.

Lemma 8: The worst case complexity of testing the
subset condition is O(n^{3(m^2)}) where n is the number of
distinct literals in the modified set And-Or^H and m is
the maximum number of rules for each literal in
And-Or^H (i.e., with that literal as the head).

Since Algorithm 2 is exponential in the number of rules
in And-Or^H, it is clearly worthwhile to reduce the size
of And-Or^H. We present an algorithm which should
greatly reduce the size of And-Or^H in most programs.

Algorithm 4

repeat
If there is no rule with head p, replace all right hand
side occurrences of p by 0.
Delete all rules with a 0 on the right hand side
until there is no change on the last iteration.
end.

We have the following simple lemmas:

Lemma 9: Algorithm 4 does not remove any rules
which can produce bindings for the head literal.

Lemma 10: Algorithm 4 is O(n^2) where n is the
number of rules.

4. Safety Analysis Using FDs and Monotonicity Con-
straints

The framework that we have developed may also be
extended to cover the use of monotonicity constraints in
the safety analysis. Let r_i and r_j be attributes of predi-
cate r. A monotonicity constraint is a couple r_i > r_j.
The constraint r_i > r_j holds in an instance of r if the
value in the i-th attribute is strictly greater than the value

335
in the jth attribute in every tuple. We may also specify
a monotonicity constraint \( r_1 > c \) (resp. \( r_1 < c \)) where \( c \) is a constant. This constraint holds in an instance of \( r \) iff
the value in the jth attribute 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

**Example 12**

Consider the predicate \( f(X,Y) \). If we define \( f(X,Y) \) by \( Y = 2 \times X \), then we have the finiteness dependencies \( f_2 \rightarrow f_1 \) and \( f_2 \rightarrow f_1 \). If we define \( f(X,Y) \) by \( Y = 2 \times X, X > 0, Y > 0 \), we have the finiteness dependencies \( f_1 \rightarrow f_2 \) and \( f_1 \rightarrow f_2 \), and the monotonicity constraint \( f_2 > f_1 \). If we define \( f(X,Y) \) by \( X < 0 \) and \( Y > 0 \), there is no finiteness dependency between the attributes 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 dependency \( f_1 \rightarrow f_2 \), but there is no monotonicity constraint

It is clear that if an argument place in a literal occurrence is determined to be safe using only FD information, additional monotonicity constraints do not affect its safety. However, certain argument places whose safety could not be detected using only the FD information might well be shown safe by using the additional information in the monotonicity constraints.

The following example illustrates how considering monotonicity constraints can allow us to detect safety in some cases where an analysis using finiteness dependencies alone fails to do so.

**Example 13**

\[
\begin{align*}
\tau(X,Y) & \leftarrow f(X,Y), g(U,V), r(Y,V) \\
r(X,U) & \leftarrow b(X,U)
\end{align*}
\]

\( b \) is a base predicate and in both the (infinite) predicates \( f \) and \( g \), the second argument finitely determines the first, \( r_1 \leftarrow f_2 \rightarrow f_1 \) and \( g_2 \leftarrow g_1 \). Consider the query \( \tau(X,Y) \). As in the first example, we can repeatedly apply the first rule (after applying the second initially), possibly generating a (finite) number of new values that satisfy \( \tau \) in each application. Given only the above FD information about \( f \), it is not possible to determine whether this process converges, and so the predicate computed for \( \tau \) is potentially infinite.

However, if we are also given the monotonicity constraints \( r_2 > c \) and \( g_2 > g_1 \), where \( c \) is a constant, we can infer that the first rule can only be applied a finite number of times (i.e., subsequent applications will not produce anything new) This ensures that both \( r_1 \) and \( r_2 \) can only receive a finite number of values and are therefore safe

It is clear that if an argument place in a literal occurrence is determined to be safe using only FD information, additional monotonicity constraints do not affect its safety. However, certain argument places whose safety could not be detected using only the FD information might well be shown safe by using the additional information in the monotonicity constraints.

In order to reason with monotonicity constraints, we need the notion of an argument mapping. Our definition is a straightforward extension of the definition presented in [Afrati et al. 86].

Consider the set of adored rules \( \mathbf{H} \). Let \( p \) and \( q \) be names of derived literal occurrences. An argument mapping \( (p, q) \) is a mixed graph with the set of nodes being the argument places of \( p \) and \( q \). For each rule, we obtain an argument mapping between each pair of derived literal occurrences in the rule (including the head literal and derived literal occurrences in the rhs). We draw an (undirected) edge \( (p_i, q_j) \) if the same variable occurs in the \( i \)th argument place of \( p \) and the \( j \)th argument place of \( q \). We draw an arc \( (p_i, q_j) \) if we are able to infer (from the monotonicity constraints in this rule) that \( X > Y \) for some variable \( X \) in the \( i \)th argument place of \( p \) and \( Y \) in the \( j \)th argument place of \( q \).

An argument mapping \( (p, q) \) can be composed to yield a composite mapping \( (p, r) \) by merging the corresponding nodes in \( q \). We note that a composite argument mapping represents the bindings involved in a sequence of rule applications. In particular, the composite mapping \( (p, q_1) \) \( (q_1, q_2) \) \( (q_n, p) \) represents a cyclic path, and we can complete the cycle by joining the corresponding nodes in the two instances of \( p \) with edges. Further, if \( q_i \leftarrow p_i \), then this is a simple cycle.

We can 'summarize' the composite mapping \( (p, r) \) into an ordinary argument mapping as follows. We draw an edge \( (p_i, r_i) \) if there is an undirected path between \( p_i \) and \( r_i \), and an arc \( (p_i, r_i) \) if there is a directed path from \( p_i \) to \( r_i \). Having done this for all pairs \( (p_i, r_i) \), we delete all nodes in \( q \) and the edges and arcs incident on them.

An argument mapping \( (p, q) \) is invalid if it contains both arcs \( (p_i, q_j) \) and \( (q_j, p_i) \). If one of the argument mappings corresponding to a rule is invalid, the rule is invalid. If the summary of a composite argument mapping is invalid, the composition is invalid. Intuitively, an invalid rule or composition cannot produce any answers. Without loss of generality, we assume that the rule does not contain any invalid rules (if it does, we simply ignore them in the safety analysis since they cannot produce any bindings).

A node in an argument mapping is bounded above (below) if every value it can receive is less than (greater than) some constant. A cycle in an argument mapping is
bounded above (below) if it contains at least one node that is bounded above (below). Clearly, a cycle is bounded above and below if it contains a safe node.

We now consider in more detail how argument mappings correspond to rule sequences. Given a sequence of rules (in $H^*$) such that the head of each rule has the same predicate name as that of a literal occurrence in the body of its predecessor, we obviously have a corresponding argument mapping. If such a sequence of rules is cyclic, we call it a rule cycle. Thus, there is a cyclic argument mapping corresponding to each rule cycle. Further, we show (Lemma 11) that there is a rule cycle corresponding to each cycle in a graph $\text{And}_H(p)$.

We define a decreasing cycle in a graph $\text{And}_H(p)$ as follows. It is a cycle which is also a directed cycle in the argument mapping for the corresponding rule cycle, and the direction of the cycle in $\text{And}_H(p)$ is opposite to its direction in the argument mapping. An increasing cycle in $\text{And}_H(p)$ is similarly defined. It is a cycle which is also a directed cycle in the argument mapping for the corresponding rule cycle, and the direction of the cycle in $\text{And}_H(p)$ is the same as its direction in the argument mapping.

We also refer to a node that is bounded above as an upper bounded node. A similar lemma holds for nodes that are bounded below (lower bounded nodes).

We now present a theorem which is a stronger version of Theorem 3.

**Theorem 5:** $p_i$ is safe if every possible subset $\text{And}_H(p)$ contains one of the following:
- a 0-node,
- a cycle containing no $f$-nodes,
- an increasing cycle with a node $q_j \preceq q_i < p_i$, and
- a cycle which can only be traversed a finite number of times.

We observe that given a graph $\text{And}_H(p)$, we can check whether it satisfies one of the conditions a), b) or c) of Theorem 5 without referring to any other graphs. Checking condition d) requires consideration of other graphs as well.

Due to space limitations, we do not present the algorithm to determine whether an argument place $p_i$ is safe. The interested reader is referred to [Ramakrishnan et al. 87] for the algorithm.

5. Safety, Finiteness of Intermediate Results, Termination

In this section, we consider the relationship between the issues of safety, finiteness of intermediate results, and termination. In addition to clarifying these issues, the discussion serves to illustrate some potential applications of the results presented in earlier sections.

A query was defined to be safe if it had a finite set of answers. Another important property of a query is whether there exists some computation that satisfies the following two conditions:

1. At any stage in the computation, each intermediate relation is finite (i.e., we only examine finite subsets of relations).
2. It enumerates the answer set.

We call this property finiteness of intermediate relations, with the implicit understanding that we refer to the existence of a computation which satisfies the above criteria.

Note that this is a property which could hold even if the answer set were infinite, since all that is required is that the intermediate relations (including the current version of the answer set as a special case) be finite at any given step, and there is no bound on the number of steps. Conversely, a query could enjoy this property and yet be unsafe. This follows immediately from the fact that the number of steps is not bounded. Further, the existence of infinite relations, including infinite base relations, does not preclude this property, since a computation might exist which only examines finite subsets...
of these relations at each step

**Termination** is usually defined to mean that the computation enumerates all answers and subsequently stops. Given this definition, the existence of a terminating computation which enumerates the answer set implies safety and finiteness of intermediate relations. However, in [Afrati et al. 86] a specific top-down algorithm is defined to terminate if it constructs a finite AND-OR tree with the leaves being base relations, finite or infinite, which collectively contain all answers to the query. This definition of termination is strictly weaker than the first, and given this definition of termination, the existence of a terminating computation no longer implies safety or finiteness of intermediate relations.

The following example illustrates the notion of termination which is used in [Afrati et al. 86].

**Example 14**

\[
\text{r}(X) - f(X)
\]

If 'f' denotes an infinite relation, 'r' is clearly unsafe, and any attempt to enumerate all X such that r(X) holds must have an infinite number of steps. The top-down algorithm described in [Afrati et al. 86], however, is defined to terminate trivially since the tree simply has a single leaf 'f' which contains all answers.

Safety is a property that is independent of the computation producing the set of answers. The issues of finite intermediate results and termination are, however, properties of the computation which produces the answer set. In order to reason about them, we need to make certain assumptions about the computation, in particular about the infinite relations. The following set of assumptions is illustrative:

1. Given an infinite relation f(X) and an element 'a', we can test if 'a' is in f (i.e., if f(a) holds) by considering a finite subset of f.
2. Given an infinite relation f(X,Y) and an element 'a', we cannot, in general, test whether there exists some Y such that f(a,Y), by considering only a finite subset of f.
3. Given an infinite relation f(X,Y) and an element 'a', if f₁ → f₂ we can test whether there exists some Y such that f(a,Y), by considering only a finite subset of f.

The generalization of these assumptions to n-ary relations and multiple finiteness dependencies is straightforward. There is nothing sacrosanct about this set of assumptions - several equally reasonable alternatives are conceivable. We would normally choose a set of assumptions that accurately reflect the actual nature of the infinite relations that we are dealing with. However, the above assumptions are intuitively appealing, and it is instructive to see what can be said about the existence of a computation with finite intermediate results and about termination, given these assumptions.

**Example 15**

Consider the following rules:

\[
\text{r}(X) - f(X,Y), \text{r}(Y) \\
\text{r}(X) - b(X)
\]

Let the query be r(X)?, let 'b' denote a finite base relation, and let 'f' denote an infinite base relation. The query is clearly unsafe, and there is no computation with finite intermediate relations which enumerates the answers. There is no terminating computation using either definition of termination. If we add the constraint f₂ → f₁, the query is still unsafe, and there is no terminating computation. However, under the given set of assumptions the bottom-up computation with sideways information passing (from r to f₂) enumerates all answers and has finite intermediate relations.

Now consider the query r(5)? The query is safe, but there is no computation which terminates (by either definition of termination) or has finite intermediate relations. If we add either the constraint f₁ → f₂ or the constraint f₂ → f₁, there exists a computation (top-down or bottom-up, depending on which constraint we add) which has finite intermediate relations and eventually establishes r(5) if it is true. However, this computation is not guaranteed to terminate in the event that r(5) is not true. If the constraint f₂ → f₁ holds, and in addition we have f₂ > f₁ or f₂ < f₁, then we can also guarantee the existence of a terminating computation with finite intermediate results which establishes r(5) if it is true. The definition of termination in [Afrati et al. 86], however, does not apply since this is a bottom-up algorithm.

We now state a result about the existence of a computation which enumerates all answers and has finite intermediate relations.

**Theorem 6**: Given a query q and a set of Horn Clauses H, there exists a computation which enumerates all answers and, at each step, examines only finite subsets of relations iff the fixpoint solution of all possible sets AND₂(pᵢ), where pᵢ is not bound by the query, assigns '0' to each qᵢ, where q is an rhs literal occurrence such that there is a variable which appears in one of its argument places and also in the argument place of another literal occurrence.

6. Conclusions

If we are to design a Horn Clause programming language which is complete in that it guarantees all answers to a given query, it is necessary that we be able
to test whether a given query has a finite set of answers, since only such queries can be allowed in the language. While the problem of detecting safety is in general undecidable, we have shown that it is decidable under a reasonable abstraction. Our abstraction, which consists of representing constants, arithmetic operations and function symbols using (possibly infinite) database relations which are not interpreted, is still sufficiently expressive to enable us to detect the safety of a large class of problems. The analysis also illustrated the relationship between the issues of safety, termination and finiteness of intermediate relations. Further, the framework allows us to reason about finiteness of intermediate relations under different assumptions about the nature of the infinite relations that represent function symbols.

7. References

[Afra et al 86]
"Convergence of Sideways Query Evaluation."

[Ramakrishnan et al 87]
"Safety of Recursive Horn Clauses With Infinite Relations," R. Ramakrishnan, P Bancilhon and A Silberschatz, Manuscript

[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

[Ullman 82]

[Ullman and Van Gelder 85]

[Zaniolo 86]