DECIDABILITY AND EXPRESSIVENESS ASPECTS
OF LOGIC QUERIES

by

Oded Shmueli

Technical Report #420
June 1986
DECIDABILITY AND EXPRESSIVENESS ASPECTS
OF LOGIC QUERIES

by

Oded Shmueli

Technical Report #420

June 1986
DECIDABILITY AND EXPRESSIVENESS ASPECTS OF LOGIC QUERIES

Oded Shmueli

Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel

Abstract

This paper addresses some basic problems regarding logic programming based queries over relational databases. Specifically, it considers the problems of determining containment, equivalence, satisfiability and expressiveness of logic queries. We re-examine the connection between the query classes $H$ and $YE^+$ defined by Chandra and Harel [2]; surprisingly, their result that $H=YE^+$ hinges on the use of the inequality operator. The containment and equivalence problems addressed here extend the work of Aho, Sagiv and Ullman on relational queries [1].

Keywords: Formal Languages, Logic Programming, Relational Databases, Queries, Undecidability

# Part of this work was conducted at the IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y. 10598, US.
1. INTRODUCTION

Recently, there has been a growing interest in logic programming based query languages and their relationship to traditional database theory [2, 9, 12] and practice [4, 5, 7, 8, 10, 12, 13]. This paper addresses some basic problems regarding logic based queries. Specifically, it considers the problems of determining containment, equivalence and satisfiability of logic queries.

It is shown that determining containment or equivalence of logic queries is recursively undecidable even when no constant or function symbols are allowed. This extends the work of Chandra and Harel on containment and equivalence of relational queries [1]. These results also apply to $YE^*$ queries [2]. We re-examine the connection between $H$ and $YE^*$ [2]; surprisingly their result that $H = YE^*$ hinges on the use of the inequality operator. We show that positive $H^*$ queries have expressive power greater than $YE^{++}$ queries (precise definitions below). It is interesting to determine whether undecidability extends to containment and equivalence of $YE^{++}$ queries.

Satisfiability is decidable for positive queries with no function symbols. Satisfiability is undecidable when function symbols are allowed.

Section 2 contains definitions. Containment and satisfiability issues are addressed in section 3. Satisfiability is treated in section 4. In section 5 it is proved that $H^*$ is more expressive than $YE^{++}$. The role of predicate arity in determining expressiveness is also explored. Section 6 presents conclusions and open problems.

2. LOGIC QUERIES

2.1 Logic Programming Concepts

We mostly follow the notation in [6]. A term is defined inductively as follows.

(1) A variable is a term.

(2) A constant is a term.

(3) If $f$ is $n$-ary function symbol and each $t_i$ is term, then $f(t_1, ..., t_n)$ is also a term.

If $p$ is $n$-ary predicate and each $t_i$ is term, then $p(t_1, ..., t_n)$ is an atom. Given $\alpha = \{Y_1 / t_1, ..., Y_n / t_n\}$ and a term or an atom $e$, simultaneously substitute term $t_i$ for variable $Y_i$ of $e$ and obtain $e'$. This is called a substitution of $e$ according to $\alpha$ and is written $e' = e_\alpha$. The $Y_i$'s are said to be bound to the $t_i$'s. Let $\alpha$ and $\beta$ be substitutions. Let $e$ be a term or an atom, then $e_\alpha \beta$ means the application of $\beta$ to $e_\alpha$. 
Substitution $\alpha$ is a unifier for $e_1$ and $e_2$ if $\delta \alpha = t \alpha$. Unifier $\alpha$ is a most general unifier (mgu) for $e_1$ and $e_2$ if for each substitution $\delta$ such that $e_1 \delta = e_2 \delta$ there exists a substitution $\beta$ such that $e_1 \alpha \beta = e_1 \delta$ and $e_2 \alpha \beta = e_2 \delta$. All mgu's for a given set of expressions are variants, i.e. identical up to variable renaming.

A clause is a formula of the form $(X_1 \lor \ldots \lor X_m)$ where each $X_i$ is a literal, i.e. either an atom or a negation of an atom. A clause can be represented as $(B_1 \land \ldots \land B_n) \rightarrow (A_1 \lor \ldots \lor A_m)$ where the $A_j$’s are the non-negated $X_i$’s (also called the clause head) and the $B_j$’s appear as negated $X_i$’s (also called the clause body). A clause with at most one non-negated literal is called a Horn clause. A goal clause is a Horn clause with all literals negated. An expression (i.e. a term, literal or clause) containing no variable occurrence is said to be ground.

A logic program is a finite set of Horn clauses. Let $P$ be a logic program and $g$ be a goal; goal $g'$ is derived from goal $g$ if**:

1. $g = A_1 \land \ldots \land A_n$;
2. $P$ contains a clause $A \leftarrow B_1 \land \ldots \land B_n$ such that $A$ and $A_1$ are unifiable with mgu $\alpha$, i.e. $\alpha \alpha = A_1 \alpha$
3. $g' = A_1 \alpha \land \ldots \land A_{i-1} \alpha \land B_1 \alpha \land \ldots \land B_n \alpha \land A_{i+1} \alpha \land \ldots \land A_n \alpha$.

We use $g \vdash g_1$ to indicate that goal $g_1$ is derived (using a program or a database clause) from goal $g$. A sequence $g_1, \ldots, g_k$ of goals where for $i=2, \ldots, k$, $g_i$ is obtained via a clause derivation from $g_{i-1}$ is called a SLD-derivation of $g_k$ from $g_1$ and is denoted by $g_1 \vdash^* g_k$.

A successful derivation, or a proof, of goal $g$ is a SLD-derivation $g \vdash^{*}$ ending with the empty clause, i.e. a SLD-refutation.

2.2 Semantics of Logic Queries

A logic query is a logic program in which predicates are implicitly partitioned into program predicates and database predicates. Program predicates are those predicates which appear in the head of some program clause. A program predicate always appears with the same arity. Database predicates are those predicates which only appear in bodies of program clauses. A query contains exactly one goal clause, called the query goal, and without loss of generality (w.l.o.g.) the query goal contains a single literal.

A logic query (or simply a query) is meant to be applied to databases. A database is a finite set of ground unit clauses (i.e. facts) over database predicates; essentially, the database is a relational database.

** We use & and $\land$ interchangeably.
with relation names corresponding to database predicates and tuples corresponding to ground unit clauses. Unless stated otherwise, all domains are assumed to be infinite and no order on domain elements is assumed.

**Example 1:** A logic query.

(goal) ← q(X,Y).

(i) q(X,Z) ← p(X,Y) & l(Y,Z).

(ii) q(X,Y) ← l(X,Y). □

In the above example there are two program clauses (i) and (ii). The only database predicate is l. The goal has two variables, in general the goal or the program clauses may include constants.

The *result* of the *application* of a logic query Q to a database D, denoted Q(D), is a set of tuples of the form (·) where q(·) is the predicate occurrence in Q's goal. A tuple (·) appears in the result with certain values replacing the query variables iff there is a SLD-refutation of q(·) in which the query variables are bound to those values. A result tuple may contain unbound variables.

**Example 2:** When the query of Example 1 is applied to

\[ D = \{l(1,2),l(2,3),l(3,1),l(4,5)\}; \]

the result is

\[ \{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3),(4,5)\}. □ \]

A query Q is *satisfiable* if there exists a database D such that Q(D) ≠ ∅. Query Q₁ contains query Q₂, written Q₁ ⊇ Q₂, if for all databases D, Q₁(D) ⊇ Q₂(D). Queries Q₁ and Q₂ are equivalent, written Q₁ = Q₂, if for all databases D, Q₁(D) = Q₂(D).

The analysis of queries is organized according to their syntax. A query is *positive* if all literals appearing in clause bodies in the query are positive. Queries are further classified into four subclasses:

- *cfff* - the query contains neither constants nor function symbols;
- *cfeb* - the query contains no constants but may include function symbols;
- *cbff* - the query may include constants but has no function symbols;
- *cbfb* - the query may contain both constants and function symbols.
3. CONTAINMENT AND EQUIVALENCE

We prove that for $cifff$ queries, determining containment or equivalence is recursively unsolvable. As a corollary, it is also recursively unsolvable to determine containment or equivalence for $cifff$, $cifb$ and $cifb$ queries. In [9] the following corollary is stated: "Telling whether two sets of positive clauses with equality compute the same mapping from database relations to defined relations is undecidable". We note that (i) positive Horn clauses in [9] include negation of database relations (predicates), (ii) the notion of what is being computed is different in [9] than the one used in this paper and (iii) no additional operators (e.g. equality or inequality) are used in this paper. Hence, the results in this section are stronger than the corollary in [9].

The proof is by reduction from decision problems for context-free languages (CFLs) [3]. Intuitively, the database is used to encode terminal symbols in grammars as well as strings (a technique identical to the one employed in [9]). A string $a_1 \ldots a_k$ is encoded by a chain of triples of the form $l(0,a_1,1), \ldots, l(k-1,a_k,k)$ where $j=0, \ldots, k$ are distinct constants (not necessarily integers) and $a_j$ are constants in 1-1 correspondence with the grammar’s set of terminal symbols. Similarly, any chain of triples, in which the $j=0, \ldots, k$ are not necessarily distinct, also represents a string.

The reduction is by associating with a context-free grammar (CFG) $G$ a logic query $Q(G)$ with query goal $g$. Suppose $G$ is given by $(N,T,P,S)$ where $N$ is a finite set of non-terminal symbols, $T = \{a_1, \ldots, a_n\}$ is a finite set of terminal symbols, $P$ is a finite set of grammar productions and $S$ is the sentence (start) symbol, $N \cap T = \emptyset$ and $S \in N$. Let $\Rightarrow$ denote a single grammar derivation and $\Rightarrow^*$ a sequence of zero or more grammar derivations.

We shall assume that (*) the grammar contains no empty productions (i.e. a production in which the empty string is produced from a non-terminal symbol), and that $S$ does not appear in the right hand side of any grammar production. Any CFG can be transformed to this form, except perhaps for a production in which the empty string is produced from $S$ [3]. So, we are only treating a subset of the CFGs.

$Q(G)$ is constructed thus. The query goal is $g(I,J)$ where both $I$ and $J$ are variables. Intuitively, the goal seeks a derivation starting with a triple whose first entry is bound to $I$ and ending with a triple whose last entry is bound to $J$. One program clause is:

$$g(I,J) \leftarrow t(J_0, A_1, J_1) \land \ldots \land t(J_{n-1}, A_n, J_n) \land z(I, J, A_1, \ldots, A_n).$$
The remaining program clauses are constructed in 1-1 correspondence with $G$'s productions. Consider a production $V \rightarrow b_1 \ldots b_m$ where $V$ is a non-terminal symbol and for $j = 1, \ldots, m$, $b_j \in (T \cup N)$. The corresponding program clause is: $v(I, A_1, \ldots, A_n) \leftarrow C_1 \& \ldots \& C_m$ where for $k = 1, \ldots, m$, if $b_k = a_k \in T$ then $C_k$ is $I(J_{k-1}, A_k, J_k)$, otherwise $b_k = W \in N$ and $C_k$ is $w(I_{k-1}, J_k, A_1, \ldots, A_n)$. For convenience, in the example below we write $I$ instead of $I_0$ and $J$ instead of $I_m$.

Example 3: Consider $G_1 = (N, T, P, S)$ with

- $T = \{a_1, a_2, a_3\}$.
- $N = \{Q, R, S\}$.
- $P = \{S \rightarrow Q, Q \rightarrow a_1 R a_2, R \rightarrow a_1 a_1 Q, R \rightarrow a_2 a_3\}$.

The corresponding query, $Q(G_1)$ is:

- $g_1(I, J)$.
- $g_1(I, J) \leftarrow t(I_0 A_1 J_1) \& t(I_1 A_2 J_2) \& t(I_2 A_3 J_3) \& s(I, J, A_1 A_2 A_3)$.
- $s(I, J, A_1 A_2 A_3) \leftarrow q(I, J, A_1 A_2 A_3)$.
- $r(I, J, A_1 A_2 A_3) \leftarrow l(I, A_1 A_2 A_3)$.
- $r(I, J, A_1 A_2 A_3) \leftarrow l(I, A_1 A_2 A_3)$.
- $r(I, J, A_1 A_2 A_3) \leftarrow l(I, A_1 A_2 A_3)$.

**Proposition 1:** Let $G$ be a CFG and $Q(G)$ (abbr. Q) the corresponding query.

(i) All $t \in Q(D)$ are ground.

(ii) If $t \in Q(D)$ then there is a SLD-derivation of $t$ with an intermediate state of the form:

- $l(I_0 A_m J_1) \& \ldots \& l(I_{m-1} A_m J_m) \& t(I_0 A_1 J_1) \& \ldots \& t(I_{n-1} A_n J_n)$.

(iii) Let $Z_0, \ldots, Z_{k-1}$ be distinct unbound variables.

- $S \Rightarrow a_1, \ldots, a_k$ iff

$g(I, J) \leftarrow l(I, A_1 Z_1) \& \ldots \& l(Z_{k-1} A_1 J_1) \& t(I_0 A_1 J_1) \& \ldots \& t(I_{n-1} A_n J_n)$.

**Proof:** (i) In a clause of $Q$, each head variable also appears in the clause body. Consider a SLD-derivation and a variable $V$ appearing in a literal at some step and not appearing in the next step. Either a literal containing $V$ the variable is replaced by other literal(s) which also contain that variable (with a new name), or such a literal is unified with a database clause in which case $V$ is bound.
(ii) By (*), a literal in a SLD-derivation is either replaced by some other literal(s) or is unified with a database clause. A "Switching Lemma" type argument (see [6]) can be used to construct a SLD-derivation with the desired form.

(iii) By induction on the length of the derivation and the 1-1 correspondence between program clauses and grammar productions.

Lemma 1: Let $G_1$ and $G_2$ be CFGs and $Q_1, Q_2$ the corresponding queries with query goals $g_1$ and $g_2$; $Q_1 \subseteq Q_2$ iff $L(G_1) \subseteq L(G_2)$.

Proof:

(\Rightarrow) Consider a string $\alpha \in L(G_1)$, $\alpha = a_{i_1} \ldots a_{i_k}$, we shall show that $\alpha \in L(G_2)$. Construct a database $D$ such that the $l$ clauses of $D$ form a triple chain of the form $l(b_0, a_{i_1}, b_1), \ldots, l(b_{k-1}, a_{i_k}, b_k)$ encoding $\alpha$ and the $t$ clauses form a triple chain $t(c_{0, a_{i_1}} c_1), \ldots, t(c_{n-1, a_{i_k}} c_n)$. (The $b_i$'s and $c_j$'s are distinct constants. The $t$ chain will force a 1-1 correspondence between variable $A_i$ and constant, i.e. terminal symbol, $a_{i_1}$.)

Consider the sequence of elementary grammar derivations within $S_1 \overset{*}{\Rightarrow} \alpha$ in $G_1$. By Proposition 1(iii),

\[(**)
\]

$g_1(i, J) \vdash l(i, A_{i_1}, J_1) \& \ldots \& l(i, A_{i_k}, J_k) \& t(J_0, A_{i_1}, J_1) \& \ldots \& t(J_{n-1}, A_{i_k}, J_n).

Clearly, we can match the right hand side conjunction of (***) with the $l$ triple chain encoding $\alpha$ in $D$ and the $t$ triple chain forcing, for $i=1, \ldots, k$, $A_i$ to be bound to $a_{i_i}$. Therefore, $D$ satisfies $g_1(i, J)$ with $l$ bound to $b_0$ and $J$ bound to $b_k$ and since $Q_1 \subseteq Q_2$, $D$ satisfies $g_2(i, J)$ with $l$ bound to $b_0$ and $J$ bound to $b_k$.

Consider a SLD-derivation sequence for $g_2(b_0, b_k)$. By Proposition 1(ii), there is an intermediate state of the form: $l(l_0, A_{m_1}, J_1) \& \ldots \& l(l_{w-1}, A_{m_w}, J_w) \& t(J_0, A_{i_1}, J_1) \& \ldots \& t(J_{n-1}, A_{i_k}, J_n)$. Continuing the refutation from this state, because of the two database chain constructed, we have: $w=k$, $J_0$ bound to $b_0$, $I_w$ bound to $b_k$ and for $j=1, \ldots, w$, $A_{m_j}$ bound to $a_{i_j}$. Because the $t$ chain forces $A_i$ to be bound to $a_i$, it follows that for $j=1, \ldots, k$, $A_{i_j} = A_{m_j}$. Thus, the intermediate state above can be written as:

$l(l_0, A_{i_1}, J_1) \& \ldots \& l(l_{k-1}, A_{i_k}, J_k) \& t(J_0, A_{i_1}, J_1) \& \ldots \& t(J_{n-1}, A_{i_k}, J_n)$. By Proposition 1(iii), $S_2 \overset{*}{\Rightarrow} a_{i_1} \ldots a_{i_k}$ in $G_2$; so $\alpha \in L(G_2)$.

(\Leftarrow) By Proposition 1(i), all result tuples are ground. Consider a database $D$ satisfying $g_1(i, j)$ for domain elements $i, j$. We have to show that $D$ satisfies $g_2(i, j)$ as well. By chain construction and Proposition 1(ii), there exists a successful SLD-derivation of $g_1(i, j)$ with an intermediate state of the form:

$g_1(i, J) \vdash l(i, A_{i_1}, J_1) \& \ldots \& l(i, A_{i_k}, J_k) \& t(J_0, A_{i_1}, J_1) \& \ldots \& t(J_{n-1}, A_{i_k}, J_n)$. By Proposition 1(iii),


\[ S_1 \implies a_{i_1} \ldots a_{i_k}. \] Since \( L(G_1) \subseteq L(G_2) \), \( S_2 \implies a_{i_1} \ldots a_{i_k}. \) By Proposition 1(iii),

\[ g_2(I, J) \models t(f, A_{i_1, J_1}) \& \ldots \& t(f, A_{i_k, J_k}) \& t(J_{0, A_{i_1, J_1}}) \& \ldots \& t(J_{n-1, A_{i_k, J_k}}). \]

Clearly, \( D \) satisfies \( g_2(i, j) \) by binding \( J \) to \( i \) and \( J \) to \( j \) and applying the same additional derivation steps as in the proof of \( g_1(i, j) \).

\[ \Box \]

**Theorem 1:** It is recursively unsolvable to determine, for arbitrary \( c f / i f f \) queries \( Q_1, Q_2 \), whether (i) \( Q_1 \subseteq Q_2 \), (ii) \( Q_1 = Q_2 \).

**Proof:** (i) It is recursively unsolvable to determine, for arbitrary CFGs \( G_1, G_2 \), whether \( L(G_1) \subseteq L(G_2) \) [3]. This holds even when \( G_1 \) and \( G_2 \) satisfy assumption (\( * \)). By Lemma 1, \( L(G_1) \subseteq L(G_2) \) iff \( Q_1 \subseteq Q_2 \) and hence determining \( Q_1 \subseteq Q_2 \) is recursively unsolvable.

(ii) It is recursively unsolvable to determine, given arbitrary CFGs \( G_1 \) and \( G_2 \) (even under (\( * \))), whether \( L(G_1) = L(G_2) \) [3]. Consider the corresponding queries \( Q_1, Q_2 \). Clearly, \( L(G_1) = L(G_2) \) iff \( Q_1 = Q_2 \). Therefore, it is recursively unsolvable to determine, given arbitrary queries \( Q_1 \) and \( Q_2 \), whether \( Q_1 = Q_2 \).

\[ \Box \]

4. SATISFIABILITY OF LOGIC QUERIES

**Lemma 2:** The problem of determining, for an arbitrary \( c b / i f f \) (or \( c f / i f f \)) positive query \( Q \), whether \( Q \) is satisfiable is solvable.

**Proof:** The problem is similar to that of deciding whether the language generated by a CFG is non-empty.

\[ \Box \]

**Theorem 2:** The problem of determining, for an arbitrary \( c b / f b \) positive query \( Q \), whether \( Q \) is satisfiable is unsolvable.

**Proof:** By reduction from the modified Post correspondence problem (MPCP). The Post correspondence problem (PCP) is as follows: given \( k \) pairs of strings \( x_1, y_1, \ldots, x_k, y_k \) determine whether there is a sequence \( i_1, \ldots, i_k \) such that \( x_1, x_2, \ldots, x_k = y_1, y_2, \ldots, y_k \), where for \( i = 1, \ldots, k \), \( x_i = x_{i_1} \ldots x_{i_k} \) and \( y_i = y_{i_1} \ldots y_{i_k} \). MPCP is PCP with the added requirement that \( i_1 = 1 \). Both PCP and MPCP are recursively unsolvable [3].

We transform an MPCP instance into a positive query. The MPCP instance contains \( k \) pairs of strings. The query contains the predicate \( \text{append}(L_1, L_2, L_3) \) used for concatenating lists. Append does not
require the use of negation. There is a predicate \( s(X_1,Y_1,X_2,Y_2) \); intuitively \( X_1 \) and \( Y_1 \) are strings "guessed" so far in a nondeterministic fashion, and \( X_2 \) and \( Y_2 \) are the results of concatenating, for some \( i \), \( x_i \) to \( X_1 \) and \( y_i \) to \( Y_1 \). One of the program clauses is \( s(X_1,X,X,X) \leftarrow dbr{e/l} \). This clause "checks" whether a solution for MPCEP has been obtained and the (only) database predicate \( dbr{e/l} \) has some database clause with which to unify. (For clarity we use \( A.B \) to abbreviate list\((A,B)\).) The query goal is:

\[
\leftarrow s(x_{11},x_{12}, \ldots , x_{k_1}, \text{bla}, y_{11}, \ldots , y_{k_1}, \text{bla} X_1, X_2),
\]

For each pair \( x_i, y_i \) there is a program clause of the form:

\[
s(X_1,Y_1,X_2,Y_2) \leftarrow \text{append}(x_{i_1}, \ldots , x_{i_1}, \text{bla} X_1,X_0)
\]

\& \append(y_{i_1}, \ldots , y_{i_1}, \text{bla} Y_1,Y_0)
\]

\& \( s(X_0,Y_0,X_2,Y_2) \).

By construction, the query constructed above is satisfiable iff there exists a solution for the MPCEP instance. This proves that satisfiability of positive \( cb/fb \) queries is recursively unsolvable. \(\square\)

Example 4: Reduction

**MPCEP instance:**

\[
\begin{align*}
x_1 &= \text{aba} & y_1 &= \text{ab} \\
x_2 &= \text{b} & y_2 &= \text{abb} \\
x_3 &= \text{baa} & y_3 &= \text{baa}
\end{align*}
\]

**Goal:**

\[
\leftarrow s(\text{list}(a,\text{list}(b,\text{list}(a,\text{bla}))),\text{list}(a,\text{list}(b,\text{bla})),X_1,X_2).
\]

**Program clauses:**

\[
s(X,X,X,X) \leftarrow dbr{e/l} \.
\]

\append(\text{bla} X,X).

\append(X,\text{bla}X).

\append(\text{list}(X,T),\text{list}(Y,S),\text{list}(X,Z)) \leftarrow \append(T,\text{list}(Y,S),Z).
\]

\[
s(X_1,Y_1,X_2,Y_2) \leftarrow \append(\text{list}(a,\text{list}(b,\text{list}(a,\text{bla}))),X_1,X_0)
\]

\& \append(\text{list}(a,\text{list}(b,\text{bla})),Y_1,Y_0)
\]

\& \( s(X_0,Y_0,X_2,Y_2) \).
9

\[ s(X_1,Y_1,X_2,Y_2) \leftarrow \text{append } (\text{list } (b, \text{list } (b, \text{bla })),X_1,X_0) \]
\& \hspace{1em} \text{append } (\text{list } (a, \text{list } (b, \text{bla })),Y_1,Y_0) \\
\& \hspace{1em} s(X_0,Y_0,X_2,Y_2). \]

\[ s(X_1,Y_1,X_2,Y_2) \leftarrow \text{append } (\text{list } (b, \text{list } (a, \text{list } (a, \text{bla }))),X_1,X_0) \]
\& \hspace{1em} \text{append } (\text{list } (b, \text{list } (a, \text{list } (a, \text{bla }))),Y_1,Y_0) \\
\& \hspace{1em} s(X_0,Y_0,X_2,Y_2). \]

As a corollary to Theorem 4 we obtain the following results.

**Lemma 3:** The problem of determining, for an arbitrary *cf ifb* positive query \( Q \), whether \( Q \) is satisfiable is unsolvable.

**Proof:** Consider an arbitrary *cb ifb* query \( Q \). Form a new query \( Q_1 \) by globally replacing each constant \( c \) appearing in \( Q \) with a term \( c(Z) \) where \( c \) is a unique function symbol and \( Z \) is a unique new variable.

Clearly, \( Q_1 \) is satisfiable iff \( Q \) is satisfiable. \( \square \)

**Corollary:** It is undecidable whether a given database \( D \) satisfies a *cb ifb* positive query \( Q \). \( \square \)

5. \( H^+ \) PROPERLY CONTAINS YE**

5.1 Containment

Chandra and Harel defined a class \( H \) of queries which is essentially the *cf iff* logic queries [2]. The semantics of \( H \) queries is similar to that of the Logic Queries defined in this paper. The main difference concerns the treatment of unbound variables. In [2] these are implicitly bound to some element in a finite set \( D \) which is part of the database definition. Equality ( = ) and Inequality ( ≠ ) are also allowed in \( H \) queries. A distinguished predicate symbol is called the *carrier* of the query; it corresponds to the predicate in the query goal in our notation.

The semantics given to \( H \) queries is a "bottom-up" iterative fixpoint computation. The computation is done in stages until convergence. A set \( T_i \) is built at stage \( i \). Set \( T_0 \) is initialized to contain all database clauses (facts). At stage \( i, i > 1 \), each program clause \( c \) is examined. If the literals in \( c \)'s body can be unified with a set of ground clauses out of \( T_{i-1} \) then \( c \)'s head, with variables replaced with the values with which they were unified, is added to \( T_i \); if some variable appearing in \( c \)'s head is not bound by the above unification then it can be replaced by any constant out of the set \( D \). The set of tuples computed are those carrier facts in \( T_n \), the set at convergence. The semantics of [2] also assumes "built-in" database relations one containing all equal pairs out of \( D \) and one containing all non-equal pairs out of \( D \).
Define $H^+$ to be $H$ with the use of inequality prohibited. Our results on the undecidability of determining containment and equivalence of logic queries carry over to $H^+$ queries. This follows from the fact that in the logic queries constructed to show undecidability we may assume $(*)$, i.e. the CFGs contain no derivation of the empty string from a non-terminal symbol; thus a successful SLD-derivation explicitly binds all variables introduced during the computation. Therefore, on this class of queries $H^+$ and the equivalent logic queries have the same semantics.

Another class of queries, $YE^+$, has also been defined in [2]; these are queries which are represented by a fixpoint applied to a positive existential query. Equality ($=$) and inequality ($\neq$) are also allowed in $YE^+$ queries. The expression $(zy.Ys)^\Psi$ represents a query, where $\Psi$ is a first order formula involving only database predicates and s, and (i) the length of z plus y, (ii) the arity of s and (iii) the number of free variables in $\Psi$ are all equal. The meaning of such a query is to first compute the least fixpoint $s=\Psi(s)$, and then obtaining tuples using $zy$ as a selector (i.e. a result tuple components must be the same if the corresponding variables in the vector $zy$ are identical). The query's result is obtained by projecting those tuples on the first $|z|$ components where $|z|$ denotes the length of $z$.

**Example 5:** The following $YE^+$ query computes the reflexive and transitive closure of a database relation $r$: $(X_1,X_2),Ys)(\exists Z)(X = Y \lor (r(X,Z) \land s(Z,Y)))$.

A $YE^+$ query is a query of the form $(zy.Ys) (\exists x)(\Phi(x,u))$, where $\Phi$ is a positive (i.e. the only boolean connectives in $\Phi$ are $\land$ and $\lor$) quantifier free boolean formula, $x$ (u) is the vector of bound (respectively free) variables in $\Phi$; and $z$ is a vector of variables not occurring in $\Phi$ whose length plus the length of $y$ is equal to the length of $u$ and the arity of $s$. Define $YE^{++}$ to be $YE^+$ with the use of inequality prohibited.

Two queries in $YE^{++}$, or two programs in $H^+$, or a query in $YE^{++}$ and a program in $H^+$, are equivalent if over all databases they compute identical results.

**Lemma 4:** A $YE^{++}$ query $Q$ is equivalent to a $H^+$ program having a single recursive program predicate symbol.

**Proof:** A method for transforming a $YE^+$ query into an equivalent $H$ program is given in [2]. Over $YE^{++}$ queries this method can be slightly modified to produce an equivalent $H^+$ program containing exactly one clause of the form: $a(X_1,\ldots,X_l) \leftarrow c(Y_1,\ldots,Y_l)$ (where $a$ is the query predicate and each $Y_i$ may be the
same as some \( X_j \). It may only contain (either directly or indirectly) recursive program clauses of the form: \( c(Y_1, \ldots, Y_j) \leftarrow q_1 & \cdots & q_n & c_1 & \cdots & c_m, n \geq 0, m \geq 0 \) (where each \( q_i \) is a database predicate and each \( c_j \) has the form \( c(Z_1, \ldots, Z_j) \)).

For a \( YE^{++} \) query \( Q \) of the form \( (X_1, X_2, \ldots, X_j) \). Ys \( \Phi \) this can be done as follows. First \( \Phi \)'s matrix is converted to disjunctive normal form (no use of negation on positive formulas is necessary). Next, for each disjunct of the form \( (A_1 \land \cdots \land A_k) \) a clause \( s(X_1, \ldots, X_j) \leftarrow A_1 & \cdots & A_k \) is formed where \( X_1, \ldots, X_j \) are the free variables in \( \Phi \). The clause \( t(X_i) \leftarrow \neg s(X_1, X_2, \ldots, X_j) \) is added where \( t \) is the program's query predicate. Finally, equality is enforced by using variable renaming instead of the equality predicate symbol.

Chandra and Harel have showed that \( H = YE^{++} \). Next, we exhibit that because the use of inequality is prohibited, \( H^+ \) has more expressive power than \( YE^{++} \). Consider the program \( P \) illustrated below. The query predicate (carrier) is \( t \).

\[
\begin{align*}
t(X) & \leftarrow b_1(X, Y) \land b_2(Y, X). \\
b_1(X, Y) & \leftarrow r_1(X, Y). \\
b_1(X, Y) & \leftarrow r_1(X, Z) \land b_1(Z, Y). \\
b_2(X, Y) & \leftarrow r_2(X, Y). \\
b_2(X, Y) & \leftarrow r_2(X, Z) \land b_2(Z, Y). 
\end{align*}
\]

(Intuitively, we think of \( r_i(a, b) \) as a graph edge directed from \( a \) to \( b \) and labeled \( i \) (i=1,2). The above query asks for graph nodes \( v \) such that the following property holds relative to \( v \) (***) there exists a graph node \( w \), a path connecting \( v \) to \( w \) whose edges are marked with 1, and a path from \( w \) to \( v \) whose edges are marked with 2.)

**Theorem 3**: \( H^+ \) properly contains \( YE^{++} \).

**Proof**: By Lemma 4 \( H^+ \) contains \( YE^{++} \). To show proper containment, it suffices to exhibit an \( H^+ \) program for which there is no equivalent \( YE^{++} \) query. We claim that \( P \) is such a program, i.e. that there exists no \( YE^{++} \) query equivalent to program \( P \).

For the sake of deriving a contradiction assume there exists a \( YE^{++} \) query \( Q \) which is equivalent to \( P \). W.l.o.g \( Q \) has the form \( (X_1, X_2, \ldots, X_j) \). Ys \( \Phi \). Transform \( Q \) using the method of Lemma 4 into an equivalent \( H^+ \) program \( P' \) with carrier \( t \). We now argue that \( P' \) cannot be equivalent to \( P \).
P' cannot have a clause of the form \( s(X_1, \ldots, X_j) \leftarrow \); otherwise it would compute incorrectly on an empty database. For P' to compute an answer in a bottom-up fashion, it must have a program clause(s) whose body is solely composed of database predicates. Let \( C \) be the set of these clauses.

Consider a clause \( c \in C \). The body of \( c \) naturally defines an edge labeled directed graph by associating with each body literal \( r_i(X,Y) \) a graph edge \((x,y)\) labeled i. The defined graph is satisfying if it satisfies property (***) relative to node \( x \). If a body of a clause in \( C \) does not define a satisfying graph relative to \( x_1 \), then we can form a database corresponding to the graph's edges on which \( P' \) would compute an answer, namely \((x_1)\), that should not appear in the result. So, the body of a clause in \( C \) must define a satisfying graph.

Since \( P' \) is a finite object, it follows that bodies in clauses of \( C \) can only define a (small) subset of the satisfying graphs (relative to \( x_1 \)). So, we can choose a sufficiently large graph \( G \) satisfying (***) relative to node \( x_1 \) in which no sub-graph is isomorphic to any satisfying graph defined by the body of any clause in \( C \). \( G \) consists of two sufficiently long \((m, \text{ the number of symbols in } P', \text{ will suffice})\) chains of the form \( r_1(x_1,a_2), \ldots, r_1(a_m,x_1), r_2(a_2,x_1), \ldots, r_2(x_1) \). On \( G \)'s database representation \( P' \) computes, erroneously, no answer tuple at all. \( \square \)

5.2 Predicate Arity and Expressiveness

The effects of predicate arity on expressiveness are considered next. First, we exhibit a program \( P_1 \) for which there is no \( YE^{++} \) equivalent query in which predicate arity equals two. For \( i=1,2 \), \( P_1 \) has the following set of clauses:

\[
\begin{align*}
& t(X,Y) \leftarrow b_i(X,Y).
& b_i(X,Y) \leftarrow r_i(X,Y).
& b_i(X,Y) \leftarrow r_i(X,Z) \land b_i(Z,Y).
\end{align*}
\]

The query predicate (carrier) is \( t \). Each \( b_i \) predicate set generates answers which are the endpoints of an \( r_i \) labeled path in the graph defined by the database.

**Lemma 5**: There exists no \( YE^{++} \) query, in which only arity two predicate symbols are used, which is equivalent to \( P_1 \).

**Proof**: For the sake of deriving a contradiction assume that there exists such a \( YE^{++} \) query \( Q \) which is equivalent to \( P_1 \). A \( H^+ \) program is minimal if (i) no program clause may be deleted and still resulting in
an equivalent program (relative to the query answers) and (ii) no group of literals in any clause may be deleted and still resulting in an equivalent program. W.l.o.g Q has the form \((X,Y) \neq \emptyset\) where the arity of s is two.

Transform Q using the method of Lemma 4 into an equivalent \(H^*\) program. Let \(P_1\) be a minimal version of that program. Considering derivations by \(P_1\) on databases which contain only \(r_1\) facts or only \(r_2\) facts, by condition (i) no clause may mention both \(r_1\) and \(r_2\) (all answers may be computed without such a clause and adding a clause cannot eliminate a previous answer). We claim: (iii) there is no program clause, with head \(s(X_1,X_2)\), in which some of the database literals in the body, w.l.o.g \(r_1\) literals, form a chain (within the clause body) "connecting" \(X_1\) to \(X_2\) which is isomorphic to \(r_1(X_1=X_1,X_1),...,r_1(X_1=X_1,X_1)\). Such a chain, by itself, derives the head; furthermore, deleting all body literals which are not in the chain results in an equivalent program violating condition (ii).

Consider any recursive program clause (one must exist in \(P_1\)) of the form:

\[ s(U,V) \leftarrow q_1 \land \ldots \land q_n \land s_1 \land \ldots \land s_m. \]

W.l.o.g the predicate symbol in all \(q_i\)'s is \(r_1\), the predicate symbol in each \(s_i\) is \(s\). We now describe a construction of a database on which \((u,v)\) can be derived erroneously. Define a directed graph \(G=(W,E)\) where \(W=\{x \mid X \text{ appears in the clause}\}\) and \(E=\{(x,y) \mid \text{for some } 1 \leq i \leq n, q_i=r_1(X,Y)\}\). Define \(R_u=\{x \mid x = u \text{ or } (x \in W \text{ and } x \text{ is reachable via a directed path from } u)\}\) define \(R_v=\{y \mid y = v \text{ or } (y \in W \text{ and there is a directed path connecting } y \text{ and } v)\}\). By (iii) there is no directed path connecting node \(u\) to node \(v\) in \(G\) (corresponding to the head variables are \(U\) and \(V\)); i.e. \(R_u \cap R_v\) is empty. Define \(N=W-(R_u \cup R_v)\).

Populate the database with \(S=\{r_1(x,y) \mid (x,y) \in E\}\); this concurrently satisfies \(q_i, 1 \leq i \leq n\). Next, \(s_1, \ldots, s_m\) are satisfied. Consider a \(s_i\) literal which has the form \(s(Z_1,Z_2)\). By condition (ii), it is impossible that \(Z_1=X\) and \(Z_2=Y\). So, we have the following cases to consider:

\[ X, Y \in R_1: \text{ Add database clause } r_1(x,y). \]

\[ X, Y \in R_2: \text{ Add database clause } r_1(x,y). \]

\[ X, Y \in N: \text{ Add database clause } r_1(x,y). \]

\[ X \in R_u, Y \in R_1: \text{ Add database clause } r_2(x,y). \]

\[ X \in R_1, Y \in R_u: \text{ Add database clause } r_1(x,y). \]

\[ X \in R_v, Y \in N: \text{ Add database clause } r_1(x,y). \]
\[ X \in N, Y \in R_u: \text{Add database clause } r_1(x, y). \]

\[ X \in N, Y \in R_v: \text{Add database clause } r_2(x, y). \]

\[ X \in N, Y \in N: \text{Add database clause } r_3(x, y). \]

Observe that \( s(Z_1, Z_2) \) can always be satisfied with a fact of the form \( r_1(x, y) \) or a fact of the form \( r_2(x, y) \). Also, by condition (ii), the body contains no literal \( s(U, V) \). By construction, all the body literals are concurrently satisfied, \( t(u, v) \) can be derived and yet there is no path connecting \( u \) and \( v \) which is made only of \( r_1 \) facts or only of \( r_2 \) facts. Thus \( Q \) cannot exist. \( \square \)

Interestingly, if predicate symbols with a larger arity are allowed, then there exists a \( YE^{++} \) query equivalent to \( P_1 \). We shall first show a \( H^+ \) program \( P_1' \), having a single recursive program predicate, which is equivalent to \( P_1 \). (We use ‘_’ to denote a unique variable name.)

\[
\begin{align*}
t(X,Y) & \leftarrow s(X,Y,\ldots,\ldots). \\
s(X,Y,X,Y,U,U) & \leftarrow r_1(X,Y). \\
s(X,Y,V,V,X,Y) & \leftarrow r_2(X,Y). \\
s(X,Y,X,Y,U,U) & \leftarrow r_1(X,Z) \land s(\ldots,Z,Y,\ldots). \\
s(X,Y,V,V,X,Y) & \leftarrow r_2(X,Z) \land s(\ldots,\ldots,Z,Y). 
\end{align*}
\]

Intuitively, the first two places in \( s \) designate correct answers, the second two places correspond to \( r_1 \) chains and the last two places correspond to \( r_2 \) chains. The idea is that even when \( s(\ldots) \) is used "incorrectly", no harm is done since in that case variable equality is forced so that the program still computes correctly. The \( YE^{++} \) query is:

\[
((X,Y),Ys)((Z,ABCD)(r_1(X,Y) \lor r_2(X,Y) \lor (r_1(X,Z) \land s(A,B,Z,Y,C,D)) \lor (r_2(X,Z) \land s(A,B,C,D,Z,Y))).
\]

6. CONCLUSIONS

We have proved that the problem of determining containment or equivalence of logic queries is recursively unsolvable. Likewise, satisfiability of logic queries containing function symbols is undecidable. When function symbols are not allowed, satisfiability is decidable. It is interesting to note that satisfiability of function free logic queries in which restricted negation (i.e. applied only to database predicates) is allowed, is also decidable [11].
Chandra and Harel have defined two query classes, $H$ and $YE^*$, and proved that these classes are identical [2]. Our containment and equivalence results extend to these classes. We have defined $H^+$ and $YE^{++}$ to be those classes with the exception that the use of inequality ($\neq$) is prohibited. We have showed that $H^+$ is more expressive than $YE^{++}$. The complexity of deciding containment or equivalence over $YE^{++}$ queries remains an open problem.

The role of predicate arity in determining expressiveness was also considered. We have exhibited a program that cannot be expressed using a single arity two recursive predicate. However, there is an equivalent program with a single, arity six, recursive predicate. This raises the (open) problem of characterizing the queries which can be expressed with a single recursive predicate.
ACKNOWLEDGMENT

I had stimulating discussions with K. R. Apt, A. K. Chandra, N. Francez, A. Itai, A. Walker and R. Y. Pinter. I would especially like to thank N. Francez for reading and commenting on early versions of this paper. None of those mentioned above is in any way responsible for the content of this paper.

REFERENCES


