EQUIVALENCE OF DATALOG QUERIES IS UNDECIDABLE*

ODED SHMUELI

▷ Datalog is a powerful query language for relational databases [10]. We consider the problems of determining containment, equivalence, and satisfiability of Datalog queries.

We show that containment and equivalence are recursively unsolvable. This should be contrasted with the work of Aho, Sagiv, and Ullman on relational queries [1]. Satisfiability is easily decidable for Datalog queries. We also consider Datalog^f which allows function symbols. Here, satisfiability is recursively unsolvable. ◁

1. INTRODUCTION

Recently, there has been a growing interest in logic-programming-based query languages and their relationship to traditional database theory [2, 5, 6, 8, 7, 10]. Datalog queries are composed of a logic program and a single goal. They are similar to Prolog programs without extralogical operators. This paper addresses some basic problems regarding Datalog queries. Specifically, it considers problems concerning containment, equivalence, and satisfiability of Datalog queries. The results of this paper were first reported in [9] within a different setting (e.g., in the definition of application of a query).

It is shown that determining containment or equivalence of Datalog queries is recursively unsolvable. This should be contrasted with the work of Aho, Sagiv, and Ullman on containment and equivalence of relational queries [1]. These results also apply to H queries [2]. The results are proved using recursively unsolvable problems from the theory of context-free languages. As a corollary, we show that literal redundancy determination is a recursively unsolvable problem.

*A preliminary version of this paper is contained in "Decidability and Expressiveness Aspects of Logic Queries," in the ACM Symp. on Principles of Database Systems, 1987.

Address correspondence to Oded Shmueli, Department of Computer Science, Technion—Israel Institute of Technology, Haifa 32000, Israel.

Received January 1991; accepted June 1992.

Papadimitriou has obtained a similar equivalence result for a restricted subset of Prolog whose syntax and semantics differ from Datalog; the result is sketched in his pioneering paper [6]. This language allows the use of negation and equality which are not part of the vocabulary of Datalog. Papadimitriou's result follows from the undecidability of equivalence of polynomial-time bounded Turing machines. Our result follows from the undecidability of equivalence in a simpler system, namely, context-free languages.

Satisfiability is decidable for Datalog queries. Satisfiability is recursively unsolvable when function symbols are allowed (the language is denoted Datalog^f). As a corollary, we show that determining whether a Datalog^f query is safe, i.e., always producing finite result sets, is recursively unsolvable.

The paper is organized as follows. Section 2 contains definitions. Containment and equivalence issues are addressed in Section 3. Satisfiability is treated in Section 4. Section 5 concludes.

2. DATALOG QUERIES

We mostly follow the notation in [4]. A *term* is defined inductively as follows. 1) A variable is a term; 2) A constant is a term; 3) If f is an n-ary function symbol and each t_i is a term, then $f(t_1, \ldots, t_n)$ is also a term. If p is an n-ary predicate symbol and each t_i is a term, then $p(t_1, \ldots, t_n)$ is an *atom*. A *literal* is either an atom or a negated atom. A *clause* is a formula of the form $(\forall \bar{X})(C_1 \vee \cdots \vee C_m)$ where each C_i is a literal and $\bar{X}$ is the sequence of variables appearing in the C_is. We usually omit the $\forall \bar{X}$ and leave it implicit when we write a clause. A clause can be represented as $(B_1 \wedge \cdots \wedge B_n) \rightarrow (A_1 \vee \cdots \vee A_m)$ where the A_js are the nonnegated C_is (also called the *clause head*) and the B_js appear as negated C_is (also called the *clause body*).

A clause with at most one nonnegated literal is called a *Horn clause*. We write Horn clauses as
Head $\leftarrow B_1, \ldots, B_m$,
where *Head* is the clause head and the B_is are the clause body. A *goal clause* (or simply a *goal*) is a Horn clause with all literals negated. A *unit clause* is a Horn clause consisting of a single nonnegated literal. The empty clause is denoted $\square$.

An *expression* is a term, a literal, or a clause; an expression containing no variable occurrence is said to be *ground*.

A substitution α is a set of pairs Y_i/t_i, $1 \le i \le n$, where $Y_1, \ldots, Y_n$ are distinct variables and $t_1, \ldots, t_n$ are terms such that Y_i is distinct from t_i, $1 \le i \le n$. Given substitution $\alpha = \{Y_1/t_1, \ldots, Y_n/t_n\}$ and an expression E, simultaneously substitute term t_i for variable Y_i of E and obtain E'. This is called a *substitution* of E according to α and is written $E' = E\alpha$. The Y_is are said to be *bound* to the t_is. Let α and β be substitutions. Let E be an expression; $E\alpha\beta$ means $(E\alpha)\beta$.

A substitution α is a *unifier* for E_1 and E_2 if $E_1\alpha = E_2\alpha$. Unifier α is a *most general unifier* (*mgu*) for E_1 and E_2 if for all substitutions γ such that $E_1\gamma = E_2\gamma$ there exists a substitution β such that $E_1\alpha\beta = E_1\gamma$ and $E_2\alpha\beta = E_2\gamma$.

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

- $g = \leftarrow A_1, \ldots, A_n$;
- P contains a clause $c = A \leftarrow B_1, \ldots, B_m$, w.l.o.g. g and c have no variables in common, such that A and some A_i are unifiable with mgu α, i.e., $A\alpha = A_i\alpha$.

(w.l.o.g., only variables appearing in g or c appear in α and if variables W, appearing in A, and V, appearing in A_i, form a pair in α, then the pair is W/V; i.e., goal variables "survive.")

- $g' = \leftarrow A_1\alpha, \ldots, A_{i-1}\alpha, B_1\alpha, \ldots, B_m\alpha, A_{i+1}\alpha, \ldots, A_n\alpha.$

When P is understood, we use $g \vdash g_1$ to indicate that goal g_1 is derived, in P, from goal g. A sequence $g_1, \ldots, g_k, k \geq 1$ of goals, where for $i = 2, \ldots, k$, g_i is obtained via a derivation from g_{i-1} using α_{i-1}, is called an *SLD-derivation* of g_k from g_1 and is denoted by $g_1 \vdash^k g_k$. $g_1 \vdash^* g_2$ indicates that $g_1 = g_2$ or for some $k > 0$, $g_1 \vdash^k g_2$.

A *successful derivation* of goal g is an SLD-derivation $g \vdash^* \square$ ending with the empty clause, also called an *SLD-refutation.*

A *Datalog query* Q (or simply a query) is of the form (P, g). P is a logic program containing neither constants nor function symbols, in which predicate symbols are partitioned into *EDB predicate symbols* and *IDB predicate symbols*. *IDB* predicate symbols are those predicate symbols which appear in the head of some program clause. *EDB* predicate symbols are those predicate symbols which only appear in bodies of program clauses. g is a goal clause called the *query goal*; g is of the form $\leftarrow A$ where A is an atom; no constants or function symbols may appear in A.

A Datalog query is meant to be applied to databases. A *database* is a finite set of ground unit clauses (also called *facts*) with *EDB* predicate symbols; essentially, the database is similar to a relational database with relation names corresponding to *EDB* predicate symbols and tuples corresponding to ground unit clauses.

Example 1. A Datalog query computing the transitive closure of l. (goal) $\leftarrow q(X, Y)$.

1) $q(X,Z) \leftarrow q(X,Y), l(Y,Z).$
2) $q(X,Y) \leftarrow l(X,Y).$ ■

In the above example, there are two program clauses 1) and 2). The only *EDB* predicate symbol is l. The goal has two variables.

Let U be a set of constants. The *result* of the *application* of a Datalog query $Q = (P, \leftarrow q)$ to a database D *modulo* U, denoted $Q(D, U)$, is $\{q\alpha_1, \ldots, \alpha_k\delta \mid$ for some k and δ, on $P \cup D$, $\leftarrow q \vdash^k \square$ with α_i used in the ith step and δ assigns to all variables in $q\alpha_1 \cdots \alpha_k$ constants from $U\}$. U is fixed as the domain out of which the database constants are taken, and we write $Q(D)$ instead of $Q(D, U)$. U may be infinite, and no order is assumed on its elements. Observe that the above definition makes $Q(D)$ a set of facts.

Example 2. When the query of Example 1 is applied to $U = \{1,2,3,4,5,\ldots,10\}$, $D = \{l(1,2), l(2,3), l(3,1), l(4,5)\}$, the result is $\{q(1,1), q(1,2), q(1,3), q(2,1), q(2,2), q(2,3), q(3,1), q(3,2), q(3,3), q(4,5)\}$. ■

A query Q is *satisfiable* if there exists a database D such that $Q(D) \neq \varnothing$. Query Q_1 *contains* query Q_2, written $Q_1 \supseteq Q_2$, if for all databases D, $Q_1(D) \supseteq Q_2(D)$. Queries Q_1 and Q_2 are *equivalent*, written $Q_1 = Q_2$, if for all databases D, $Q_1(D) = Q_2(D)$. Database D *satisfies* query Q if $Q(D)$ is not empty. A query is *safe* if for all databases D, $Q(D)$ is finite.

At this point, we should clarify the relationship between Datalog queries as defined here and their more standard definition as in [10]. The syntax of Datalog as defined here is almost identical to that in [10]. One exception is that query goals may contain no constants. The results concerning containment, equivalence, and satisfiability of Datalog queries remain valid when constants are allowed in query goals.

The semantics of Datalog is traditionally defined "bottom-up," using a T_P type operator (see [4]). The set U in the definition of $Q(D,U)$ is the database domain in the Datalog definition in [10]. In computing bottom-up, we only use substitutions which assign values out of U. From the set of facts generated by this bottom-up computation, those facts unifying with the query goal atom are selected as the answer set.

In performing SLD resolution on $P \cup D$, and substituting values out of U for unbound variables following the SLD-refutation, we compute the same answer set as in the bottom-up computation. This hinges on two facts: 1) The success set of a definite program P is equal to its least Herbrand model which equals $T_P \uparrow \omega$ (Theorems 6.5 and 8.3 in [4]); and 2) Let q be an atom and $A \in B_P$, A unifies with q and $\leftarrow A$ has an SLD-refutation iff $\leftarrow q$ has an SLD-refutation with computed answer α, and there exists a substitution β such that $A = q\alpha\beta$ (Lemma 8.2 and Theorem 8.4 in [4]).

3. CONTAINMENT AND EQUIVALENCE

We prove that for Datalog queries, determining containment or equivalence is recursively unsolvable. 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 [6] and similar to that in [11]). First, U includes all of the grammar's terminal symbols, as well as other constants. A string $a_{i_1} \cdots a_{i_k}$ can be encoded by a chain of triples of the form $l(c_0, a_{i_1}, c_1) \ldots l(c_{k-1}, a_{i_k}, c_k)$, where for $j = 0, \ldots, k$, c_j are constants. If the c_js are distinct, then the encoding is called *standard*, and otherwise it is said to be *nonstandard.*

The reduction is by associating with a context-free grammar (CFG) G a Datalog query $Q(G) = (P_G, \leftarrow g(I,J))$, where P_G is a logic program which depends on G, g is a symbol not appearing in G, and I and J are variables. Suppose G is given by (N, T, P, S), where N is a finite set of nonterminal 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, let $\Rightarrow^k$ denote a length k 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 nonterminal symbol), and that S does not appear on 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; however, containment and equivalence are still recursively unsolvable on this subset.

$Q(G)$ is constructed thus. The query goal is $\leftarrow g(I,J)$. 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 (this is similar to the technique in [11]). The only program clause in P_G with head predicate symbol g is

$g(I,J) \leftarrow h(A_1,\dots,A_n), s(I,J,A_1,\dots,A_n)$.

The remaining program clauses in P_G are constructed in 1–1 correspondence with G's productions. Consider a production $V \Rightarrow b_1 \cdots b_m$, where V is a nonterminal symbol and for $j = 1,\dots,m, b_j \in (T \cup N)$. The corresponding program clause is: $v(I_0, I_m, A_1,\dots,A_n) \leftarrow C_1,\dots,C_m$, where for $k = 1,\dots,m$, if $b_k = a_j \in T$, then C_k is $l(I_{k-1}, A_j, I_k)$; otherwise, $b_k = W \in N$ and C_k is $w(I_{k-1}, I_k, A_1,\dots,A_n)$.

Example 3. Consider $G_1 = (N,T,P,S)$ with $T = \{a_1,a_2,a_3\}$, $N = \{Q,R,S\}$, $P = \{S \Rightarrow Q,\ Q \Rightarrow a_1Ra_2,\ R \Rightarrow a_1a_1Q,\ R \Rightarrow a_2a_3\}$.

The corresponding query $Q(G_1)$ is

$(\{g_1(I,J) \leftarrow h(A_1,A_2,A_3), s(I,J,A_1,A_2,A_3).$
$s(I_0,I_1,A_1,A_2,A_3) \leftarrow q(I_0,I_1,A_1,A_2,A_3).$
$q(I_0,I_3,A_1,A_2,A_3) \leftarrow l(I_0,A_1,I_1), r(I_1,I_2,A_1,A_2,A_3), l(I_2,A_2,I_3).$
$r(I_0,I_3,A_1,A_2,A_3) \leftarrow l(I_0,A_1,I_1), l(I_1,A_1,I_2), q(I_2,I_3,A_1,A_2,A_3).$
$r(I_0,I_2,A_1,A_2,A_3) \leftarrow l(I_0,A_2,I_1), l(I_1,A_3,I_2)\},$
(query goal) $\leftarrow g_1(I,J).)$. ■

Remark (global variables). Usually, the scope of a variable is limited to a single clause. For the above construction, an SLD derivation will force all references to an *IDB* literal to use the same $A_1,\dots,A_n$ as an argument. Thus, we may think of $A_1,\dots,A_n$ as being *global variables*.

Let G be a CFG and $Q(G)$ the corresponding query; let D be a database for $Q(G)$.

Proposition 1. If $t \in Q(G)(D)$, then there is an SLD-derivation of t with an intermediate state of the form (where for $1 \le u \le w$, $1 \le j_u \le n$)

$$\leftarrow h(A_1,\dots,A_n), l(I_0, A_{j_1}, I_1),\dots, l(I_{w-1}, A_{j_w}, I_w).$$

PROOF. In each literal, except for h, there are two places denoting its "before" and "after" in a "chain"; for l, these are the first and last places, whereas for nonterminal symbols, these are the first two places. Each intermediate goal can thus be looked upon as a "chain." By (*), a literal in an SLD-derivation is either replaced by some other literal(s) or is unified with a database clause. In case of replacement by other literals, the chain property of the new goal is maintained. This is ensured by the fact that each program clause "expands" a piece of the chain corresponding to a nonterminal symbol. By the Switching Lemma (see [4]), all unifications with database clauses can be done as the final steps. Thus, an SLD-derivation with the desired form can be obtained. ■

Proposition 2. Let $Z_0,\dots,Z_k$ be distinct variables; let $W \in N$.

$W \Rightarrow^* a_{i_1},\dots,a_{i_k}$

iff

$$(\leftarrow w(I,J,A_1,\dots,A_n)) \vdash^* \big(\leftarrow l(I = Z_0, A_{i_1}, Z_1),\dots, l(Z_{k-1}, A_{i_k}, Z_k = J)\big).$$

PROOF. By induction on the length of the derivation. The proof is based on the 1–1 correspondence between program clauses and grammar productions and the global variables remark. ■

Proposition 3. Let $Z_0, \ldots, Z_k$ be distinct variables.

$$S \Rightarrow^* a_{i_1}, \ldots, a_{i_k}$$

iff

$$(\leftarrow g(I,J)) \vdash^* \big(\leftarrow h(A_1, \ldots, A_n), l(I = Z_0, A_{i_1}, Z_1), \ldots, l(Z_{k-1}, A_{i_k}, Z_k = J)\big).$$

PROOF. A direct corollary of Proposition 2. ■

Lemma 1. Let G_1 and G_2 be CFGs, and let $Q(G_1)$ and $Q(G_2)$ be the corresponding queries with query goals $\leftarrow g_1(I,J)$ and $\leftarrow g_2(I,J)$; $Q(G_1) \subseteq Q(G_2)$ iff $L(G_1) \subseteq L(G_2)$.

PROOF. (only if) Consider a string $\alpha \in L(G_1)$, where $\alpha = a_{i_1}, \ldots, a_{i_k}$; we shall show that $\alpha \in L(G_2)$. Construct a database D containing a single h clause $h(a_1, \ldots, a_n)$ and l clauses which form a triple chain $l(b_0, a_{i_1}, b_1), \ldots, l(b_{k-1}, a_{i_k}, b_k)$ which is a standard encoding of α. (The b_is are distinct constants. The h clause will force a 1–1 correspondence between variable A_i and constant, i.e., terminal symbol, a_i.) Consider the sequence of elementary grammar derivations within $S_1 \Rightarrow^* \alpha$ in G_1. By Proposition 3, (**) $(\leftarrow g_1(I,J)) \vdash^* (\leftarrow h(A_1, \ldots, A_n), l(I, A_{i_1}, I_1), \ldots, l(I_{k-1}, A_{i_k}, J))$.

Clearly, we can match the right-hand side conjunction of (**) with the l triple chain encoding α in D and the h clause forcing, for $i = 1, \ldots, k$, A_i to be bound to a_i. Therefore, D satisfies $g_1(I,J)$ with I bound to b_0 and J bound to b_k, and since $Q(G_1) \subseteq Q(G_2)$, D satisfies $g_2(I,J)$ with I bound to b_0 and J bound to b_k. Consider an SLD-derivation sequence starting with $\leftarrow g_2(I,J)$. By Proposition 1, there is an intermediate state of the form (where for $1 \le u \le w$, $1 \le j_u \le n$)

$$\leftarrow h(A_1, \ldots, A_n), l(I_0, A_{j_1}, I_1), \ldots, l(I_{w-1}, A_{j_w}, I_w).$$

Continuing the derivation from this state, because of the single database chain in D, we have: $w = k$, I_0 bound to b_0, I_w bound to b_k, and for $m = 1, \ldots, w = k$, A_{j_m} bound to a_{i_m}. However, $h(A_1, \ldots, A_n)$ and the single h database clause, namely, $h(a_1, \ldots, a_n)$, force that A_u is bound to a_v iff $u = v$, so A_{j_m} is bound to a_{i_m} iff $j_m = i_m$, $1 \le m \le k$. Thus, the intermediate state above can be written as

$$\leftarrow h(A_1, \ldots, A_n), l(I_0, A_{i_1}, I_1), \ldots, l(I_{k-1}, A_{i_k}, I_k).$$

By Proposition 3, $S_2 \Rightarrow^* a_{i_1} \cdots a_{i_k}$ in G_2; so $\alpha \in L(G_2)$.
(if) Consider $(i,j) \in Q(G_1)(D)$; we have to show $(i,j) \in Q(G_2)(D)$. By Proposition 1, there exists a successful SLD-derivation starting with

$$\leftarrow g_1(I,J)$$

with an intermediate state s of the form (where for $1 \le u \le w$, $1 \le j_u \le n$)

$$\leftarrow h(A_1, \ldots, A_n), l(I_0, A_{j_1}, I_1), \ldots, l(I_{w-1}, A_{j_w}, I_w).$$

By Proposition 3, $S_1 \Rightarrow^* a_{j_1} \cdots a_{j_w}$. Since $L(G_1) \subseteq L(G_2)$, $S_2 \Rightarrow^* a_{j_1} \cdots a_{j_w}$. By Proposition 3,

$$(\leftarrow g_2(I,J)) \vdash^* (\leftarrow h(A_1,\dots,A_n), l(I, A_{j_1}, I_1), \dots, l(I_{w-1}, A_{j_w}, J)).$$

Clearly, $(\leftarrow g_2(I,J)) \vdash^* \square$ by applying the same additional derivation steps from the intermediate state s in the refutation of $\leftarrow g_1(I,J)$ above. Hence, $(i,j) \in Q(G_2)(D)$. ■

We thus obtain the following Theorem.

Theorem 1. It is recursively unsolvable to determine, for arbitrary Datalog queries Q_1, Q_2, whether (i) $Q_1 \subseteq Q_2$, (ii) $Q_1 = Q_2$.

PROOF.

1) 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 $Q(G_1) \subseteq Q(G_2)$, and hence determining $Q(G_1) \subseteq Q(G_2)$ is recursively unsolvable. Therefore, it is recursively unsolvable to determine, given arbitrary queries Q_1 and Q_2, whether $Q_1 \subseteq Q_2$.
2) 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 $Q(G_1), Q(G_2)$. Clearly, $L(G_1) = L(G_2)$ iff $Q(G_1) = Q(G_2)$. Therefore, it is recursively unsolvable to determine, given arbitrary queries Q_1 and Q_2, whether $Q_1 = Q_2$. ■

The results on the undecidability of determining containment and equivalence of Datalog queries carry over to H queries [2].

The following has been observed by Gaifman (see [7]).

Corollary. The problem of determining, for an arbitrary Datalog query Q, whether Q is equivalent to Q', where Q' is obtained from Q by removing a literal in a clause of Q, is recursively unsolvable.

PROOF. Let G_1 and G_2 be CFGs. Consider $Q(G_1)$ and $Q(G_2)$, with query goals $\leftarrow g_1(I,J)$ and $\leftarrow g_2(I,J)$. Consider query Q_u formed by the union of the clauses of $Q(G_1)$ and $Q(G_2)$ with a goal clause referencing a new clause whose body is $g_1(I,J), g_2(I,J)$. Let Q' be obtained from Q_u by removing $g_2(I,J)$ from the new clause. Clearly, Q_u is equivalent to Q' iff $Q(G_1) \subseteq Q(G_2)$. ■

For clauses with no built-in predicates, define *a safe clause* to be one in which each variable appearing in the clause head also appears in the clause body [10]. For a Datalog query $Q = (P, \leftarrow q)$ such that all clauses in program P are safe, for all databases D, $Q(D)$ is finite and $P \cup D$ has a finite Herbrand model. Safe clauses are important for practical reasons. If we "revaluate" $P \cup D$ bottom-up, each evaluation phase results in finitely many facts whose constants, inductively, are taken from D; since the database is finite, we only need a finite number of evaluation phases to compute the query's result.

Consider the construction of $Q(G)$. If for each grammar production we also add $h(A_1,\dots,A_n)$ to the constructed clause body, the result is a program whose clauses are all safe. Furthermore, this $h(A_1,\dots,A_n)$ addition results in a query $Q'(G)$

such that $Q'(G) = Q(G)$. It follows that Theorem 1 holds for the case where the logic programs of Datalog queries are constrained to be sets of safe clauses.

In [6], 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 recursively unsolvable." We note that: 1) positive Horn clauses in [6] are a subset of Prolog which allows negation of *EDB* relations (i.e., *EDB* negated literals), 2) the notion of what is being computed in [6] is different from the one used in this paper, and 3) no additional operators (e.g., inequality) are used in this paper.

4. SATISFIABILITY OF DATALOG^f QUERIES

Lemma 2. The problem of determining, for an arbitrary Datalog query Q, whether Q is satisfiable is solvable.

PROOF. Suppose a Datalog query $Q = (P, \leftarrow q)$ is satisfiable on database D. This means that there is an SLD-refutation $\leftarrow q \vdash^* \Box$ on $P \cup D$. Let D' be a database in which for each *EDB* predicate symbol r there is exactly one fact $r(a, \ldots, a)$ where a is a constant. Consider using in a refutation of $\leftarrow q$ in $P \cup D'$ the same program clauses, in the same order, as used in the SLD-refutation of $\leftarrow q$ in $P \cup D$; of course, for a database clause used in the refutation on D, there is exactly one database clause to be used in the refutation on D'. All unifications would succeed as there is only one constant symbol. We conclude that Q is satisfied in D'. So, Q is satisfiable iff Q is satisfied by D'.

In checking whether Q is satisfiable on D', we may eliminate constant a in D' and all variables in P and q, and consider all predicate symbols as 0-ary. This is because the unifications will always succeed. So, we end up with a CFG G in which the *EDB* predicate symbols in P are terminal symbols in G and the *IDB* ones are nonterminal symbols in G. Q is satisfiable iff the language generated by G is nonempty. Hence, the problem is that of deciding whether the language generated by a CFG is nonempty. This problem is decidable [3]. ■

The same result holds for Datalog^c which allows constants in both the program and goal components of a query. Let Q be a query in Datalog^c. Let us modify the proof of Lemma 2. In the proof, D' is populated so that for each *EDB* predicate symbol r, all possible facts using constants from P or q are present. Choose a constant b appearing in either P or q. If a database clause used in a refutation on D includes new constants not in P or q, then in the corresponding refutation step on D', the D' clause used will be the one in which the constant b appears in the positions where such new constants appear. So, Q is satisfiable iff Q is satisfied by D'.

The CFG construction of Lemma 2 no longer applies in the presence of constant symbols. To check satisfiability in D', it suffices to evaluate, bottom-up, $P \cup D'$ with U being the constants in D'; this evaluation terminates after finitely many phases as U is finite. Q is satisfiable iff q is unifiable with any of the resulting facts having the predicate of q.

Define a *Datalog^f* query in the same way a Datalog query is defined, except that both the logic program component and the goal component of the query may contain arbitrary terms involving function symbols (and constants). This language is powerful enough to express the partial recursive functions [4]. Hence, it is not surprising that satisfiability of queries becomes undecidable. We show this with a direct simple reduction from a well-known undecidable problem.

Theorem 2. The problem of determining, for an arbitrary Datalog^f query Q, whether Q is satisfiable is recursively unsolvable.

PROOF. By reduction from the modified Post correspondence problem (MPCP). The Post correspondence problem (PCP) is as follows: given k pairs of strings $x_1, y_1, \dots, x_k, y_k$, determine whether there is a sequence $i_1, \dots, i_n$, for $1 \leq j \leq n$, $1 \leq i_j \leq k$, such that $x_{i_1} x_{i_2} \cdots x_{i_n} = y_{i_1} y_{i_2} \cdots y_{i_n}$ (each side of this equation denotes the concatenation of n strings). MPCP is PCP with the added requirement that $i_1 = 1$. Both PCP and MPCP are recursively unsolvable [3].

Let, for $i = 1, \dots, k$, $x_i = x_{i,1} \cdots x_{i,g_i}$ and $y_i = y_{i,1} \cdots y_{i,h_i}$. We transform an MPCP instance into a Datalog query. The MPCP instance contains k pairs of strings. The query contains the 3-ary predicate symbol *append* used for concatenating lists. There is a 2-ary predicate symbol s. Intuitively, in atom $s(X_1, Y_1)$, X_1 and Y_1 are strings guessed so far in a nondeterministic fashion. *nil* is a constant (0-ary function symbol).

The query goal is [for clarity, we use $A.B$ to abbreviate list(A, B)]

$\leftarrow s(x_{1,1} \cdots x_{1,g_1}.nil, y_{1,1} \cdots y_{1,h_1}.nil)$.

For each pair x_i, y_i, there is a program clause of the form

$$\begin{aligned} s(X_1, Y_1) \leftarrow\ & append(X_1, x_{i,1} \cdots x_{i,g_i}.nil, X_2), \\ & append(Y_1, y_{i,1} \cdots y_{i,h_i}.nil, Y_2), \\ & s(X_2, Y_2). \end{aligned}$$

Intuitively, the above clause concatenates string x_i to X_1 and y_i to Y_1 and calls s recursively with the new guesses X_2 and Y_2 of MPCP solution prefixes.

One of the program clauses is

$s(X, X) \leftarrow dbrel(Y)$.

This clause checks whether a solution for MPCP has been obtained, and the literal *dbrel*(Y), where *dbrel* is the only *EDB* predicate symbol, has some database clause with which to unify.

By construction, the query constructed above is satisfiable iff there exists a solution for the MPCP instance. This proves that satisfiability of Datalog^f queries is recursively unsolvable. ■

Example 4: Reduction

MPCP instance:
$x_1 = aba\ y_1 = ab;\ x_2 = bb\ y_2 = abb;\ x_3 = baa\ y_3 = baa;$
Goal:
$\leftarrow s(list(a, list(b, list(a, nil))), list(a, list(b, nil)))$.

Program clauses:
$s(X, X) \leftarrow dbrel(Y)$.
$append(nil, X, X)$.
$append(X, nil, X)$.
$append(list(X, T), list(Y, S), list(X, Z)) \leftarrow$
$\qquad append(T, list(Y, S), Z)$.

$s(X_1, Y_1) \leftarrow$
$\quad append(X_1, list(a, list(b, list(a, nil))), X_2),$
$\quad append(Y_1, list(a, list(b, nil)), Y_2),$
$\quad s(X_2, Y_2).$
$s(X_1, Y_1) \leftarrow$
$\quad append(X_1, list(b, list(b, nil)), X_2),$
$\quad append(Y_1, list(a, list(b, list(b, nil))), Y_2),$
$\quad s(X_2, Y_2).$
$s(X_1, Y_1) \leftarrow$
$\quad append(X_1, list(b, list(a, list(a, nil))), X_2),$
$\quad append(Y_1, list(b, list(a, list(a, nil))), Y_2),$
$\quad s(X_2, Y_2).$ ■

As a corollary to Theorem 2, we obtain the following results:

Corollary. It is recursively unsolvable whether a given database D satisfies a Datalog^f query Q. ■

Corollary. The problem of determining, for an arbitrary Datalog^f query Q, whether Q is safe is recursively unsolvable.

PROOF. By reducing the satisfiability problem to the safety problem. Construct a program as in the satisfiability construction (proof of Theorem 2). Add to it the following clauses:

$g(X_1, X_2, X_3) \leftarrow s(X_1, X_2), gen(X_3).$
$gen(list(X, nil)) \leftarrow dbrel(X).$
$gen(list(X, Y)) \leftarrow dbrel(X), gen(Y).$

Form a new query goal:
$\leftarrow g(x_{11} \cdots x_{1g_1}.nil, y_{11} \cdots y_{1h_1}.nil, X_1).$
If the MPCP instance has a solution, this query, on a database in which *dbrel* is not empty, returns an infinite number of facts whose last component contains terms of the form *list*(A, B), and otherwise it returns no facts at all. Therefore, the new query is safe iff the MPCP has no solution. ■

5. CONCLUSIONS

We have proved that the problem of determining containment or equivalence of Datalog queries is recursively unsolvable. Our containment and equivalence results extend to H [2]. Likewise, satisfiability of Datalog^f queries is recursively unsolvable. Satisfiability of Datalog queries and Datalog^c queries is decidable.

I had stimulating discussions with K. R. Apt, C. Beeri, A. K. Chandra, N. Francez, A. Itai, A. Walker, and R. Y. Pinter. I would also like to thank the referees of this journal for their comments and suggestions. This work was supported by the Fund for the Promotion of Research at the Technion.

REFERENCES

1. Aho, A. V., Sagiv, Y., and Ullman, J. D., Equivalence Among Relational Expressions, *SIAM J. Computing*, 8(2):218–246 (May 1979).

2. Chandra, A. K. and Harel, D., Horn Clauses Queries and Generalizations, *J. Logic Programming*, 1:1–15 (1985).
3. Harrison, M. A., *Introduction to Formal Language Theory*, Addison-Wesley, 1978.
4. Lloyd, J. W., *Foundations of Logic Programming*, 2nd edition, Springer-Verlag, 1987.
5. Naqvi, S. and Tsur, S., *A Logical Language for Data and Knowledge Bases*, Computer Science Press, Potomac, MD, 1989 (North-Holland, 1974).
6. Papadimitriou, C. H., A Note on the Expressive Power of Prolog, *Bulletin EATCS* (June 1985).
7. Sagiv, Y., Optimizing Datalog Programs, in: *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, San Diego, CA, Mar. 1987, pp. 349–362.
8. *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1986–1990.
9. Shmueli, O., Decidability and Expressiveness Aspects of Logic Queires, in: *Proc. in ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, San Diego, CA, Mar. 1987, pp. 237–249.
10. Ullman, J. D., *Principles of Database and Knowledge-Based Systems, Vol. 1, 2*, Computer Science Press, Potomac, MD, 1988.
11. Ullman, J. D. and Van Gelder, A., Parallel Complexity of Logical Query Programs, in: *IEEE FOCS Conf.*, pp. 438–454.