Equivalence, Query-Reachability, and Satisfiability in Datalog Extensions

Extended Abstract

Alon Y. Levy*
Stanford University
levy@cs.stanford.edu

Inderpal Singh Mumick
AT&T Bell Laboratories
mumick@research.att.com

Yehoshua Sagiv†
Hebrew University
sagiv@cs.huji.ac.il

Oded Shmueli‡
Technion—Israel Institute of Technology
oshmu@cs.technion.ac.il

Abstract

We consider the problems of equivalence, satisfiability and query-reachability for datalog programs with negation and dense-order constraints. These problems are important for optimizing datalog programs. We show that both query-reachability and satisfiability are decidable for programs with stratified negation provided that negation is applied only to EDB predicates or that all EDB predicates are unary. In the latter case, we show that equivalence is also decidable. The algorithms we present are also used to push constraints from a given query to the EDB predicates. Finally, we show that satisfiability is undecidable for datalog programs with unary IDB predicates, stratified negation and the interpreted predicate $\neq$.

1 Introduction

We consider the problems of equivalence, query reachability and satisfiability in datalog programs with negation and dense-order constraints. Two programs are said to be equivalent if they produce the same answer for any given database. A predicate in a datalog program is said to be satisfiable if, for some database, the program computes a non-empty relation for it. An atom is said to be query-reachable if there is some database in which it is part of a derivation of the query predicate. These problems are important in the context of query optimization, since solving them can detect extraneous rules and predicates. We show that these problems are closely related. Specifically, satisfiability and query-reachability are approximately at the same level of difficulty, while equivalence is harder than them.

In previous work, satisfiability was shown decidable for pure datalog programs [Shm87] and decidability for datalog with dense-order constraints follows from the work of [KKR90]. Query-reachability was shown decidable for datalog with dense-order constraints [LS92].

In this paper, we show that both satisfiability and query-reachability are decidable for datalog programs with dense-order constraints and safe stratified negation provided that

1. negation is applied only to EDB predicates, or
2. all EDB predicates are unary.

We also give a new undecidability result for satisfiability that implies a new undecidability result for both query-reachability and equivalence. These undecidability results apply to datalog programs with unary IDB predicates, negation, and the interpreted predicate $\neq$. In contrast, note that equivalence is decidable for datalog programs with only unary IDB predicates (and neither negation nor interpreted predicates) [CGKV88].

In proving some of the decidability results, we use a rather powerful tool, the *query-tree,* which was first used in [LS92]. A query-tree is a finite structure that encodes all symbolic derivation trees (of a datalog program P) that satisfy some given property. Query-trees can be used not just for decision problems, but also for analyzing datalog programs and transforming those programs into more efficient ones. For example, using query-trees, it is rather easy to push the tightest possible constraints to the EDB, and to use constraints most efficiently during a magic-set evaluation. So, in essence, our results do not merely provide new decidable cases, but also imply how to push constraints in those cases. Query-trees can be viewed as a refinement of tree automata techniques for the special purpose of representing symbolic derivation trees of datalog programs. The importance of tree-automata techniques for decision problems of datalog programs was shown

*The work of this author was supported by NASA Grant NCC2-537.

†Part of the work of this author was done while visiting Stanford University, where it was supported by ARO grant DAAL03-91-G-0177, NSF grants IRI-90-16358 and IRI-91-16646, and a grant of Mitsubishi Corp.

‡The work of this author was done at AT&T Bell Laboratories.

in [Var89]. Conceivably the results we present could be obtained by applying tree-automata techniques directly, but doing so would be considerably more intricate.

2 Satisfiability, Query-Reachability and Equivalence

In this paper we consider datalog programs that may have safe stratified negation and dense-order constraints (i.e., the interpreted predicates $\neq$, $=$, $<$, and $\leq$). Note that $=$ is naturally expressed in datalog by using multiple occurrences of the same variable.

A datalog program P has IDB and EDB predicates, and one of the IDB predicates is the *query predicate.* The *result* of a datalog program P for an EDB D, denoted $P(D)$, is the relation computed for the query predicate when P is applied to D. It is convenient to assume that distinct programs have disjoint sets of IDB predicates. We denote programs by upper case letters (possibly with a subscript); the corresponding lower case letter denotes the program's query predicate (e.g., P_1 and p_1).

We investigate four properties of datalog programs: satisfiability, query-reachability, equivalence and containment.

Definition 2.1 (Satisfiability): An IDB predicate s of a program P is *satisfiable* if there is some EDB D, such that P defines a nonempty relation for s. □

Definition 2.2 (Query-Reachability): Consider an atom $q(\alpha_1, \ldots, \alpha_n)$ where q is either an EDB or IDB predicate and each α_i is either a variable or a constant. The atom $q(\alpha_1, \ldots, \alpha_n)$ is *query-reachable* with respect to program P if for some EDB D, there is a derivation tree d of a fact for the query predicate p, such that the derivation tree d has a positive ground atom that matches $q(\alpha_1, \ldots, \alpha_n)$. □

Definition 2.3 (Containment and Equivalence): A program P_1 is *contained* in a program P_2, written $P_1 \subseteq P_2$, if for all EDBs D, $P_1(D) \subseteq P_2(D)$. Programs P_1 and P_2 are *equivalent,* written $P_1 \equiv P_2$, if $P_1 \subseteq P_2$ and $P_2 \subseteq P_1$. □

The problems of equivalence, containment, satisfiability and query-reachability are closely related. The rest of this section investigates the relations between the problems, summarized in Figure 1.

Containment and Equivalence

Proposition 2.1 *An instance $P_1 \subseteq P_2$ of the containment problem can be reduced to an instance $P_1 \equiv P_2$ of the equivalence problem without introducing negation or increasing the arity of predicates.*

An instance $P_1 \equiv P_2$ of the equivalence problem can be reduced to (1) an instance $P_1 \subseteq P_2$ of the containment problem by either increasing the arity of predicates or introducing negation, or (2) two instances $P_1 \subseteq P_2$ and $P_2 \subseteq P_1$ of the containment problem. □

Satisfiability and Query-Reachability

Proposition 2.2 *An instance of the satisfiability problem can be reduced to an instance of the query-reachability problem without introducing new rules or predicates.* □

Lemma 2.3 *An instance of the query-reachability problem can be reduced to an instance of the satisfiability problem by doubling the arity of the predicates.* □

Equivalence and Satisfiability

The following relates the satisfiability problem to the complement of the equivalence and containment problems ($P_1 \not\equiv P_2$ and $P_1 \not\subseteq P_2$).

Proposition 2.4 *An instance of the satisfiability problem can be reduced to either an instance of the form $P_1 \not\subseteq P_2$ or an instance of the form $P_1 \not\equiv P_2$ by introducing one new trivial rule.* □

Lemma 2.5 *An instance $P_1 \not\subseteq P_2$ can be reduced to an instance of the satisfiability problem by adding rules that apply negation just once to a 0-arity predicate and do not increase the arity of predicates.* □

Finally, since it follows from [Shm87, UV88] that both containment and equivalence are undecidable for datalog programs with binary IDB predicates (and no negation or interpreted predicates), we get the following.

Corollary 2.1 *Query-reachability and satisfiability are undecidable for datalog programs with binary (and 0-arity) IDB predicates and stratified negation; a single occurrence of negation, in a rule of the form $q :- q_1, \neg q_2$ (where q, q_1, and q_2 are 0-arity predicates) is sufficient for undecidability.* □

In Section 5 we prove a result that implies the following theorem.

Theorem 2.2 *Containment, equivalence, query reachability, and satisfiability are all undecidable for datalog programs with unary (and 0-arity) IDB predicates, negation on nonrecursive predicates, and $\neq$. The usage of $\neq$ can be eliminated by introducing a pair of binary nonrecursive IDB predicates.* □

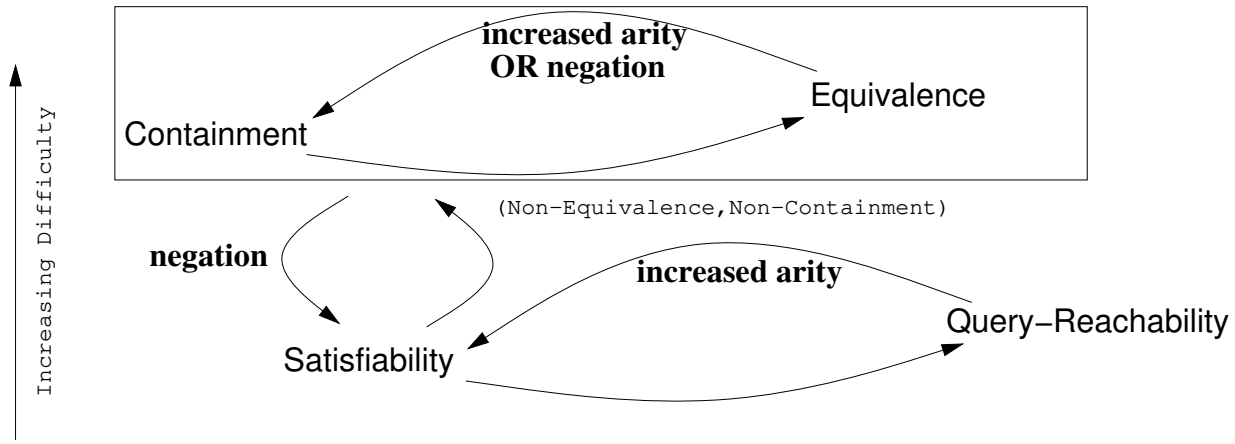

Figure 1: Relationship between satisfiability, query-reachability, containment, and equivalence

3 Datalog with Negated EDB Subgoals

In this section we show that query-reachability is decidable for datalog programs with safe negated EDB subgoals. In proving this result we use a powerful tool, the *query-tree*, first introduced in [LS92]. Query-trees are used to decide query-reachability by finitely encoding all the possible *symbolic derivations* of the query.

3.1 Symbolic Derivation Trees

A *symbolic derivation tree* for a given datalog program P (with safe negation on the EDB) may have both variables and constants and it consists of goal-nodes and rule-nodes. The only child of a goal-node g is a rule-node r, such that r is an instance of a rule of P that has g as its head. The variables of r that do not appear in g may only appear in the subtree of r. The children of a rule-node r are goal-nodes consisting of the (possibly negated) subgoals in the body of rule r. The root is a goal-node. The leaves are (possibly negated) subgoals of EDB predicates (or interpreted predicates). If a symbolic derivation tree has no variables, we call it a *ground derivation tree*, or just a *derivation tree.* If a given EDB D satisfies all the EDB subgoals of a derivation tree d, then we say that d is a derivation tree of program P with respect to the EDB D.

A symbolic derivation tree t is *satisfiable* if, by substituting constants for variables, t can be transformed into a derivation tree of P with respect to some EDB D. Note that if P has neither negation nor interpreted predicates, then any symbolic derivation tree of P is satisfiable.

EXAMPLE 3.1 Consider program P_1:

$(T1)$: $p(X,Y) :- e(X,Y).$
$(T2)$: $p(X,Y) :- e(X,Z), \neg g(X), \neg g(Z), p(Z,Y).$
$(T3)$: $c(X,Y) :- p(X,Y), g(Y), \neg e(X,Y).$

Rules $T1$ and $T2$ define a predicate p (path) in terms of EDB predicates e (edge) and g (good nodes). Rule $T3$ defines a predicate c (connectivity). Figure 2 shows one of the symbolic derivation trees t for program P_1. □

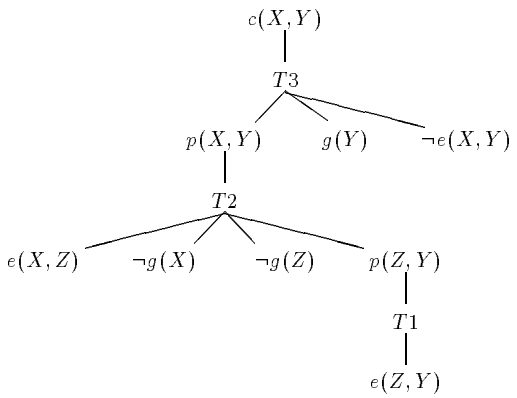

Figure 2: Symbolic derivation tree t for program P_1.

To solve the satisfiability problem for a predicate p, we should find (at least) one symbolic derivation tree of p that is satisfiable. Similarly, a given atom is query-reachable if and only if it can match some node in some satisfiable symbolic derivation tree of the query.

Unfortunately, in general there are infinitely many symbolic derivation trees. A query-tree is a *finite* structure that *encodes* a set of symbolic derivation trees.

A query-tree is useful if it encodes all and only those symbolic derivation trees (of a given program P) that satisfy a certain property. For example, given a datalog program P with negated EDB subgoals, we would like to create a query-tree T that encodes all the satisfiable symbolic derivation trees of program P.

3.2 Finite Labeling

In order to encode an infinite set of derivations in a finite structure, we attach one of a finite number of labels to every node that may appear in a derivation tree.

Consider some property Π defined on symbolic derivation trees of a given program P. We limit the discussion to properties that can be recognized by *finite labeling,* which means the following:

- There are labels (of finite size) that can be attached to nodes of a symbolic derivation tree t. The number of distinct labels is finite; the number depends on program P, but does not depend on the size of tree t.
- The label of a node is determined from the labels of its children (or vice versa).
- Whether a symbolic derivation tree t satisfies property Π can be determined from the labels of t.

Essentially, the finite labeling condition means that the set of symbolic derivation trees satisfying property Π can be recognized by a finite tree automaton.

In order to decide query-reachability of programs with negated EDB subgoals, we are interested in the property of satisfiability of symbolic derivation trees. To show that this property can be identified by finite labeling, we use *EDB labels* for goal-nodes and rule-nodes. An EDB label is either a set of EDB atoms or the *inconsistent* label, and is defined as follows.

Definition 3.1 (EDB Labels): Let t be a symbolic derivation tree of a program P. The EDB label of a rule-node r is determined from the subgoals of r as follows. Let S be the set of all EDB literals that appear either as subgoals of r or in EDB labels of IDB subgoals of r. We say that the set S is *consistent* if no IDB subgoal of r has the inconsistent label and S does not have both an atom A and its negation $\neg A$. If S is consistent, then the EDB label of rule-node r is (g, E), where g is the head of rule r and E is the set of literals of S limited to those that have only variables from g or constants. If S is not consistent, then the rule-node r gets the inconsistent label.

The EDB label of an IDB goal-node g is the same as the EDB label of its child rule node. The EDB label of an EDB goal-node e is simply $(e, \{e\})$ and will usually be omitted. □

The following proposition gives another characterization of EDB labels.

Proposition 3.1 *Consider a goal-node g in some symbolic derivation tree t. Let E be the set of all EDB literals l, such that l appears in the subtree rooted at g and has only variables of g. The EDB label of goal-node g is the pair (g, E) if E is consistent (i.e., does not include an atom A and its negation $\neg A$); otherwise, g has the inconsistent label.* □

The following proposition shows that satisfiability of a given symbolic derivation tree t can be determined just by examining the EDB labels of t.

Proposition 3.2 *A symbolic derivation tree t is satisfiable if and only if no rule-node has the inconsistent label.* □

Proof: First, note that a symbolic derivation tree is satisfiable if and only if it does not contain both an atom A and its negation $\neg A$.

For the "only if" part, suppose that t is satisfiable. Therefore, t cannot have both an EDB atom A and its negation $\neg A$. By Proposition 3.1, no goal-node, and thereby no rule-node, in t can have the inconsistent label.

For the other direction, suppose that t is not satisfiable; that is, t has both an EDB atom A and it negation $\neg A$. Consider the rule-node r which is the least common ancestor of A and $\neg A$. Rule-node r must have all the variables that appear in A and $\neg A$ and, moreover, each one of A and $\neg A$ appears either as an EDB subgoal of r or in the EDB label of some IDB subgoal of r. Therefore, r must have the inconsistent label. □

EXAMPLE 3.2 Figure 3 labels the IDB goal nodes in the symbolic derivation tree t of Example 3.1 with EDB labels. We have omitted the goal component in the EDB label shown as the goal is evident from the associated goal node. □

The only problem with the EDB labeling presented above is that it does not satisfy the first property of finite labeling, since the number of EDB labels depends on the number of variables in tree t (which depends on the size of t). Fortunately, we can consider two labels as identical when they are *isomorphic,* as defined below. We will later show that there are finitely many non-isomorphic EDB labels.

Two goal-nodes g_1 and g_2 (of the same predicate) are isomorphic if one is a variable renaming of the other, i.e., if there is a one-to-one mapping ψ from the variables of g_1 to those of g_2, such that $\psi(g_1) = g_2$. Two labels (g_1, L_1) and (g_2, L_2) are isomorphic if g_1 and g_2 are isomorphic and $\psi(L_1) = L_2$.

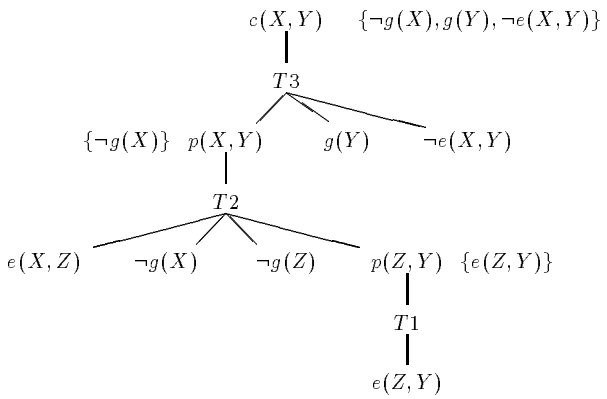

Figure 3: Labeled symbolic derivation tree t for program P_1.

3.3 The Query-Tree

As stated, a query-tree is a finite structure that encodes all the symbolic derivation trees that satisfy a certain property. A query tree is a rule-goal tree [Ull89] that is made finite by not expanding some IDB goal-nodes. Recall that a rule-goal tree is an AND-OR tree consisting of goal-nodes (the OR nodes) and rule-nodes (the AND nodes). A goal-node, g, has a child for every rule whose head unifies with g, and the actual child is the rule resulting from the unification with g. A rule-node has a goal-node child for each subgoal in its body. The root of the tree is a goal-node for the query predicate.[1] Rule-goal trees are usually infinite. We convert a rule-goal tree into a finite structure by not expanding some goal-nodes of IDB predicates (note that an *unexpanded* goal-node is a leaf). A goal-node, g, is not expanded if the query-tree already has an expanded goal-node g' that is *similar* to g. Similarity among goal-nodes is based on attaching labels to nodes and checking if two labels are isomorphic (as defined at end of Section 3.2). Formally, goal-nodes g' and g are *similar* if the labels of g and g' are isomorphic.

During the construction of the query-tree we pay special attention to the inconsistent label. Specifically, a rule-node is not made a child of a goal-node g if (after unifying r and g) r gets the inconsistent label. Of course, whether a rule-node gets an inconsistent label depends on the particular definition of labels at hand.

3.4 Encoding Symbolic Derivation Trees in the Query-Tree

The query tree encodes a set of symbolic derivations:

Definition 3.2 (Tree Encoding): Let T be a query tree for program P. A symbolic derivation tree t for program P is *encoded* in the query-tree T if there is a mapping ψ from the nodes of t to the nodes of T that satisfies the following conditions:

1. $\psi(root(t)) = root(T)$.
2. If $g_1, \ldots, g_n$ are the children of a rule-node r in t, then $\psi(g_1), \ldots, \psi(g_n)$ are the children of $\psi(r)$, respectively.
3. If a rule-node r is the child of a goal-node g in t, then $\psi(r)$ is a rule-node of the same rule as r and is a child of either $\psi(g)$ or a node g' that is similar to $\psi(g)$.
4. ψ maps each goal node of the symbolic derivation tree t to a *similar* goal node of the query-tree T.

□

To build a query-tree we need to specify how to determine the labels attached to the nodes in the tree. The main difficulty in doing so is that the tree is expanded top-down, but in general, the label of a node may depend on labels of nodes that have not yet been expanded. For example, the EDB label of a node is really determined from the labels of its children and not from the label of its parent. This presents a problem, because we need to know the label before we decide whether to expand the node. It is usually possible to solve this problem naively by introducing sufficiently many labels, but there could be more efficient solutions, as illustrated in the next section.

Therefore, we need *some* method for assigning labels to nodes of the query-tree, and that method does not always follow immediately from the basic definition of attaching labels to nodes of a symbolic derivation tree. In the next section, we explain this method for the case of the satisfiability property.

Finally, given the method for assigning labels, we must show that the query-tree encodes exactly the set of symbolic derivations satisfying a specific property Π. This means that we need to show two things. First, that every symbolic derivation satisfying Π is encoded in the query-tree. Second, we need to show that every symbolic derivation tree encoded by the query-tree satisfies Π.

3.5 Creating a Query-Tree with EDB Labels

In our discussion, we assume that no positive subgoal or head of rule in P has the same variable in two or more columns. Programs that do not satisfy this assumption can be converted into such a form using dense-order constraints discussed in the next subsection. In order to build a query-tree with EDB labels, we preface the top-down expansion of the tree by a bottom-up phase. In this phase, we compute all the possible EDB labels that may occur in derivation trees of the given datalog

[1] The query-tree could actually be a forest, where each root in the forest has one possible label of the query predicate.

program P; we do so by applying P bottom-up to an abstract interpretation in which the relation for each predicate is a set of labels. We then create *labeled* rules in which every literal is associated with an EDB label.

Step 1: (Bottom-up computation) Initially, each IDB predicate has an empty relation. In each step of the bottom-up computation, a rule r is chosen and a label is substituted for each IDB atom in the body of r. The label (s', G) can be substituted for an IDB subgoal s if there is an isomorphism ψ such that $\psi(s') = s$.

Let r be the rule

$$h \quad :- \quad s_1,\ s_2,\ \ldots,\ s_n.$$

and suppose that for $i = 1, 2, \ldots, n$, if s_i is an IDB subgoal, then the label (s_i', G_i') is substituted for s_i, where ψ_i is the isomorphism $\psi_i(s_i') = s_i$. We define $G_i = \psi_i(G_i')$ if s_i is an IDB subgoal, and $G_i = \{s_i\}$ if s_i is an EDB subgoal. Let $G_h = \cup_{i=1}^{n} G_i$. If G_h is consistent, i.e., does not have both an atom A and its negation $\neg A$, then the label (h, S) is associated with the head h, where S is the subset of all literals of G_h that have only variables from h or constants. The label (h, S) is added to the relation for the predicate of h if (h, S) is not isomorphic to an EDB label that already belongs to that relation.

EXAMPLE 3.3 Consider program P_1 from Example 3.1. We construct EDB labels for predicates p and c. Rule $T1$ derives the EDB label $(p(X,Y), \{e(X,Y)\})$. Using the EDB label $(p(X,Y), \{e(X,Y)\})$, rule $T2$ derives the EDB label $(p(X,Y), \{\neg g(X)\})$. Using the EDB label $(p(X,Y), \{\neg g(X)\})$ in rule $T2$ derives the isomorphic label $(p(X,Y), \{\neg g(X)\})$. Thus, no more EDB labels for p can be derived. Using the EDB label $(p(X,Y), \{\neg g(X)\})$, rule $T3$ derives the EDB label $(c(X,Y), \{\neg g(X), g(Y), \neg e(X,Y)\})$. Using the EDB label $(p(X,Y), \{e(X,Y)\})$ in rule $T3$ generates an inconsistency, and no new EDB label is derived. □

Step 2: (Create labeled rules) In this step, we convert the rules of P to a new set P' of *labeled* (or adorned) rules as follows. For each rule $r \in P$, we consider all possible combinations of substituting EDB labels (from the set constructed in Step 1) for the IDB subgoals of r. Note that only combinations that make r consistent are considered, and for each combination we get an EDB label for the head (the head label must also be in the set of EDB labels constructed in Step 1). The result is a new set of rules, called *labeled* (adorned) rules, so that in each rule, there is an EDB label (adornment) for the head and for each IDB subgoal. In fact, we have partitioned each IDB predicate p into several new predicates, according to the possible EDB labels for p. Note that two isomorphic EDB labels are the *same* predicate name.

EXAMPLE 3.4 Using EDB labels constructed for program P_1 in Example 3.3, we obtain the following labeled rules. Note that, to avoid ambiguity, we use the variables X_1 and X_2 in the EDB labels to represent the first and second arguments of predicates.

$$T1' : p^{(p(X_1,X_2),\{e(X_1,X_2)\})}(X,Y) \text{ :- } e(X,Y)$$
$$T2'_a : p^{(p(X_1,X_2),\{\neg g(X_1)\})}(X,Y) \text{ :- } e(X,Y), \neg g(X), \neg g(Z), p^{(p(X_1,X_2),\{e(X_1,X_2)\})}(Z,Y)$$
$$T2'_b : p^{(p(X_1,X_2),\{\neg g(X_1)\})}(X,Y) \text{ :- } e(X,Y), \neg g(X), \neg g(Z), p^{(p(X_1,X_2),\{\neg g(X_1)\})}(Z,Y)$$
$$T3' : c^{(c(X_1,X_2),\{\neg g(X_1), g(X_2), \neg e(X_1,X_2)\})}(X,Y) \text{ :- } p^{(p(X_1,X_2),\{\neg g(X_1)\})}(X,Y), g(Y), \neg e(X,Y)$$

□

Step 3: (Top-down expansion) In this step, we build the query-tree (or forest of trees[2]), in a top-down fashion, by building a rule-goal tree for the rules of P'. The EDB labels are now part of the predicate names, and therefore, two nodes are similar if they have the same predicate name (up to isomorphism). In building the tree, we only expand a goal-node if no other node in the tree with the same predicate has been expanded. Note that the inconsistent label does not appear in the tree, because all the labeled rules have consistent labels.

Step 4: (Shaking the tree) We remove from the tree all the nodes that cannot appear in derivation trees. For a node to be in a derivation tree, it must be reachable from the EDB nodes and from the root. We determine these nodes in two phases. In the first phase, we mark nodes as *reachable* until no new nodes are marked:

- All EDB nodes are marked as *reachable*
- If all the children of a rule node r are *reachable* we mark r and its parent goal-node as *reachable*.
- If g and g_1 are goal nodes of the same predicate and g is *reachable*, we mark g_1 as *reachable*.

In the second phase, we mark those nodes that are reachable from a root in the forest (until no new nodes are marked):

- If g is a root and is *reachable*, we mark it as *accessible*.
- If a goal-node g is *accessible* and r is a *reachable* child of g, we mark r and its children as *accessible*.
- If g and g_1 are goal nodes of the same predicate and g is *accessible*, we mark g_1 as *accessible*.

We remove from the query-tree all nodes that are not marked *accessible*.

Step 4 has the following effect:

[2]The forest contains a root for every EDB label that was computed in Step 1 for the query predicate.

Proposition 3.3 *Every node in the query-tree is part of a symbolic derivation encoded by the tree.* □

EXAMPLE 3.5 Figure 4 shows the query-tree built from the labeled rules constructed for program P_1 in Example 3.4. For clarity, we have omitted the goal component from EDB labels, since the goal is evident from the associated goal-node. Note that we do not expand the rightmost node $p^{\{\neg g(X_1)\}}(Z,Y)$, since there is an expanded node of that predicate. □

3.6 Proof of Correctness

We begin by showing that the number of non-isomorphic EDB labels is finite, and therefore, the construction of the query-tree terminates.

Lemma 3.4 *Consider a datalog program P with negated EDB subgoals, where a is the maximal arity of predicates, b is the maximal number of subgoals in the body of any rule, r is the number of rules, e is the number of EDB predicates, i is the number of IDB predicates, and c is the number of constants in program P.*

The maximal number of EDB labels is $l = i(a+c)^a 2^{2e(a+c)^a}$. The bottom-up computation of the EDB labels as well as the construction of the query-tree each takes $O(2be(a+c)^a r l^{b+1})$ time. □

The following lemma establishes the key property of Step 1 of the algorithm.

Lemma 3.5 *The EDB label (g, E) appears in some satisfiable symbolic derivation tree of program P if and only if Step 1 of the algorithm creates the label (g, E).* □

We now show that the query tree constructed by our algorithm encodes exactly the set of all satisfiable derivation trees.

Lemma 3.6 *Every satisfiable symbolic derivation tree of a program P is encoded by the query-tree T of P:*

Given a satisfiable symbolic derivation tree t of P, there is an encoding-mapping ψ from the nodes of t to the nodes of T that satisfies Definition 3.2. □

Proof: Let t be a satisfiable symbolic derivation tree. By Lemma 3.5, the first step of the algorithm will create a predicate $p^{(p,E)}$ where (p, E) is the label of the root of t, and therefore, $p^{(p,E)}$ will be a root in the forest.

Let g be a goal node in t for which there exists a node $\psi(g)$ in T such that the predicate of $\psi(g)$ is $g^{(g,L)}$, where (g, L) is isomorphic to the label of g. We show that the same holds for g's child rule-node r and its subgoals $g_1, \ldots, g_n$. By Lemma 3.5, Step 1 created the refined predicates $g_1^{(g_1,L_1)}, \ldots, g_n^{(g_n,L_n)}$, where (g_i, L_i) is isomorphic to the label of g_i in t. Consequently, Step 2 of the algorithm will create a refined rule

(r'): $\quad g^{(g,L)} :- g_1^{(g_1,L_1)}, \ldots, g_n^{(g_n,L_n)}$.

Note that the head of the rule *must* be $g^{(g,L)}$. Consequently, either $\psi(g)$ or some other node with the same predicate in T will be expanded with this rule. The mapping ψ will map r to r', and the children of r to the children of r' respectively. Finally, note that for all nodes $n \in t$, $\psi(n)$ will be marked as *accessible* by Step 4 of the algorithm, and will therefore remain in T. □

Lemma 3.7 *Every symbolic derivation tree encoded by the query-tree T of a program P is satisfiable:*

If a symbolic derivation tree t is encoded in the query-tree T (according to Definition 3.2), then t is a satisfiable symbolic derivation tree of P. □

Proof: It suffices to note the following: If ψ maps a goal-node $g \in t$ to a goal-node with a predicate q^L, then Condition 4 of Definition 3.2 implies that L is isomorphic to the EDB label of g in t. Since the query-tree does not contain the *inconsistent* label, it follows from Proposition 3.2 that t is satisfiable. □

From these lemmas we can decide query-reachability as follows.

Theorem 3.1 *Let T be the query-tree built for a program P.*

1. *An atom appearing in T is query-reachable with respect to P.*
2. *An atom that does not match any positive goal-node in T is not query-reachable with respect to P.*

□

Proof: Part 1 follows from Proposition 3.3 and Lemma 3.7. Part 2 follows from Lemma 3.6. □

Note that Theorem 3.1 does not say anything about atoms that do not appear in the tree, but may be obtained from atoms in the query-tree by equating two (or more variables).[3] For example, the atom $q(X,Y)$ may appear in the tree, but the atom $q(X,X)$ may not be query-reachable. To check query-reachability of such atoms, we do the following.

Let $q(\bar{X})$ be an atom that matches $q(\bar{Y})$ but is not isomorphic to it (i.e., $\bar{X}$ contains more equalities than $\bar{Y}$). We create a new predicate q' which is a *specialization* of q to the variable pattern $\bar{X}$. The arity of q' is the number of distinct variables in the pattern $\bar{X}$. We add rules for q' to P as follows. First, we add the rule

$q'(\bar{Z}) :- q(\bar{X})$.

where $\bar{Z}$ is a tuple containing the distinct variables of $\bar{X}$. In addition, for every occurrence q_0 of q in the body

[3] Or by equating a variable and a constant that appears in P.

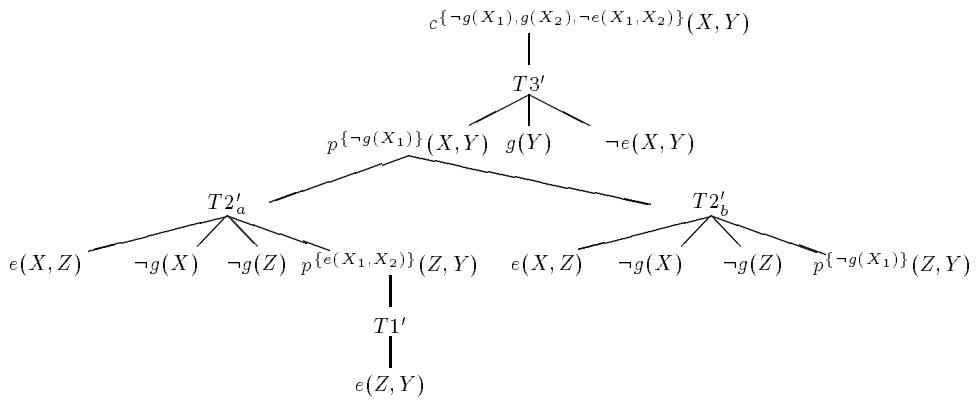

Figure 4: The query-tree built for the program P_1.

of a rule $r \in P$, add a rule r' in which the variables of r are unified with $\bar{X}$ and q_0 is replaced by q' with the appropriate variable pattern. We then build a query-tree for the resulting program, and $q(\bar{X})$ is query-reachable if and only if some atom of q' appears in the resulting query-tree. For the complete details, see [Levy93].

As a corollary of the above theorems, we get the following:

Corollary 3.2 *Given a datalog program P with negated EDB subgoals, the following problems are decidable. First, testing satisfiability of a predicate q. Second, determining whether $q(\alpha_1, \ldots, \alpha_n)$ is query-reachable with respect to P. The running time in each case is the same as that of building the query-tree, i.e., linear in the number of rules, doubly exponential in the maximal arity and singly exponential in the maximal number of subgoals (in any rule) and constants.* □

Note that the query-tree is not needed for checking satisfiability. Specifically, a predicate q is satisfiable if and only if some EDB label is generated for q during Step 1 of the algorithm. However, the query-tree is needed for query-reachability (and for pushing constraints to the EDB).

3.7 Programs With Dense-order Constraints

The algorithm, as described so far will not operate correctly on programs with dense-order constraints. The following example illustrates the problem:

EXAMPLE 3.6 Consider the following rules, where s is the query predicate:

(r_1): $p(X,Y) :- e_1(X,Y),\ e_2(Z),\ X \le Z \le Y.$
(r_2): $q(X,Y,U) :- e_3(X,Y,U),\ \neg e_2(U),\ X \ge U \ge Y.$
(r_3): $s(X,Y) :- p(X,Y),\ q(X,Y,U).$

The EDB labels would be constructed as follows. Using r_1, we first create a label $(p(X,Y), \{e_1(X,Y)\})$. Note that $e_2(Z)$ is not included in the EDB-label because Z does not appear in the head of rule r_1 nor is it known to be equal to one of the variables in the head of rule r_1. Next, with rule r_2 we will create a label $(q(X,Y,U), \{e_3(X,Y,U), \neg e_2(U)\})$. Finally, with rule r_3 we will create a label $(s(X,Y), \{e_1(X,Y)\})$. However, the constraints in the rules imply that $X = Y$, and therefore $U = Z$. Consequently, the symbolic derivation of s is inconsistent because it contains both $e_2(Z)$ and $\neg e_2(U)$. However, the EDB labels constructed were not able to detect this. □

The source of the problem is that in the bottom-up construction of the EDB labels we have made an assumption that a variable that does not appear in the head of the rule (or equal to one of the variables appearing in the head) cannot become equal to it as a result of unifications or constraint propagation higher up in the tree. However, when we have dense-order constraints, this may not be the case.

Fortunately, there is an easy fix for this problem. We begin by building a query-tree with *constraint-labels* as described in [LS92]. This results in an equivalent program P^c with rules that are refined by constraint labels. These rules will have the property that in building symbolic derivations, we do not perform non-trivial unifications, nor will we need to propagate constraints through the rules. Consequently, the same algorithm we describe in Section 3.5 will be correct when we use the rules of P^c. In our example, the resulting program P^c would be:

(r_1): $p^{X_1=X_2}(X,X) :- e_1^{X_1=X_2}(X,X),\ e_2(X).$
(r_2): $q^{X_1=X_2=X_3}(X,X,X) :- e_3^{X_1=X_2=X_3}(X,X,X),\ \neg e_2(X).$
(r_3): $s^{X_1=X_2}(X,X) :- p^{X_1=X_2}(X,X),\ q^{X_1=X_2=X_3}(X,X,X).$

and therefore the EDB label created by rule r_3 would be the *inconsistent* label.

Consequently, we get the following result (for full details, see [Levy93]):

Theorem 3.3 *Satisfiability and query-reachability are decidable for programs with dense-order constraints and negated EDB subgoals.* □

4 Programs With Unary EDBs

In this section, we consider programs in which all EDB predicates are unary. We show that these programs are bounded (even when they include stratified negation of IDB predicates and dense-order constraints). We also show that equivalence (containment), query-reachability, and satisfiability are all decidable for these programs. For simplicity, we first consider the case of programs with no constraints.

Consider a datalog program P with unary EDB predicates $e_1, \ldots, e_n$ and safe stratified negation (of EDB or IDB predicates). We will construct an equivalent non-recursive program P_1. Intuitively, the construction starts by dividing the domain of the EDB into *regions.* Since the EDB predicates are unary, each e_i forms a *basic region.* Additional regions are formed by closing the e_i's under intersection, union, and difference; thus, a region is of the form

$$\alpha_1(X),\ \ldots,\ \alpha_n(X)$$

where $\alpha_i(X)$ is a literal (i.e., an atom or a negated atom) of an EDB predicate. Note that there are only finitely many regions.

Consider a predicate q and suppose, for example, that q has two arguments. In order to identify argument positions of q easily, we use the atom $q(X, Y)$ as a template of q (i.e., X denotes the first argument position, etc.). A *generalized tuple* for $q(X, Y)$ is a conjunction of EDB literals that has (at least) one occurrence of each variable from the template $q(X, Y)$; for example,

$$e_1(X),\ e_2(X),\ \neg e_3(X),\ e_3(Y),\ e_2(U),\ \neg e_1(U),\ e_4(V)$$

is a generalized tuple. A generalized tuple defines a set of ordinary (i.e., ground) tuples as a cartesian product of regions. For example, in the above generalized tuple, the first argument position of $q(X, Y)$ is the region $e_1(X) \wedge e_2(X) \wedge \neg e_3(X)$, the second argument position is the region $e_3(Y)$, and both $e_2(U) \wedge \neg e_1(U)$ and $e_4(V)$ must be nonempty regions if the generalized tuple is to represent a nonempty set of tuples.

Note that a generalized tuple must include all the variables from the template $q(X, Y)$ and, in addition, may contain any number of other variables. However, there are only finitely many regions and, therefore, there are only finitely many nonequivalent generalized tuples.

Recall that P denotes a datalog program with unary EDB predicates $e_1, \ldots, e_n$ and safe stratified negation (of EDB or IDB predicates). We claim that the relation for each IDB predicate of P can be defined by a set of generalized tuples. In order to illustrate the proof, we evaluate P on an abstract interpretation of generalized tuples. The evaluation starts with the IDB predicates of the lowest stratum. Specifically, let r be a rule for such an IDB predicate (note that only EDB predicates may appear in negated subgoals of r). If we substitute a generalized tuple for each IDB subgoal of r, then the body becomes a conjunction of EDB literals and, hence, defines a generalized tuple.

The bottom-up evaluation terminates, since there are only finitely many nonequivalent generalized tuples. Note that a generalized tuple generated during bottom-up evaluation is added to the relation for the corresponding predicate if it is not equivalent to any generalized tuple that is already in that relation (also note that it is easy to check equivalence of generalized tuples).

After evaluating the lowest stratum, we rewrite the rules for higher strata to eliminate the IDB predicates of the lowest stratum by substituting the computed generalized tuples for subgoals of these IDB predicates. Note that when a generalized tuple is substituted for a negated subgoal, the result is not necessarily an ordinary rule (i.e., the body is not necessarily a conjunction of literals). However, the result can be converted easily to an equivalent form consisting of several rules that are ordinary ones, but with the possible exception that they may have negated subgoals of the form $\neg\exists Z e_i(Z)$. This does not present a problem, since we may view $\exists Z e_i(Z)$ as a new EDB predicate, e_i', that denotes the projection of e_i on the empty set of argument positions (and, hence, its arity is 0).

EXAMPLE 4.1 Suppose we have the following rules for the query predicate q:

$$\begin{aligned} p(X) &:- e_1(X),\ e_2(Z). \\ q(X) &:- e_3(X),\ \neg p(X). \\ q(X) &:- e_2(X),\ \neg p(X). \end{aligned}$$

The first rule generates the generalized tuple $e_1(X) \wedge e_2(Z)$ for p.

This generalized tuple is substituted for p in the rules for q, resulting in the following rules.

$$\begin{aligned} q(X) &:- e_3(X),\ \neg(e_1(X),\ e_2(Z)). \\ q(X) &:- e_2(X),\ \neg(e_1(X),\ e_2(Z)). \end{aligned}$$

The above rules can be converted to the following rules.

$$\begin{aligned} q(X) &:- e_3(X),\ \neg e_1(X). \\ q(X) &:- e_3(X),\ \neg\exists Z e_2(Z). \end{aligned}$$

$$q(X) \quad :- \quad e_2(X),\ \neg e_1(X).$$
$$q(X) \quad :- \quad e_2(X),\ \neg\exists Z e_2(Z).$$

However, the fourth rule requires that e_2 be both empty and nonempty and is, therefore, unsatisfiable. □

Theorem 4.1 *Let P be a datalog program with unary EDB predicates $e_1, \ldots, e_n$ and safe stratified negation (of EDB or IDB predicates). Program P is equivalent to a program P_1 that contains the rules $e'_i :- e_i(X)$ $(1 \leq i \leq n)$, as well as rules that have only the predicates $e_1, \ldots, e_n$ and $e'_1, \ldots, e'_n$ in their bodies.* □

Note that a rule of P_1 is unsatisfiable if it contains an atom and its negation or an atom $e_i(X)$ and the negated atom $\neg e'_i$. Unsatisfiable rules are eliminated from P_1.

Corollary 4.2 *Satisfiability, equivalence, and query-reachability are decidable for programs with stratified negation and unary EDB predicates.* □

Proof: A predicate $q \in P$ is satisfiable if and only if there is a rule in P_1 with head q (we assume that unsatisfiable rules have been deleted from P_1). In the full paper, we will show how decidability of query-reachability follows from Lemma 2.3 and its proof. Finally, two programs are equivalent if they compute the same set of generalized tuples. □

We can also prove a result similar to Corollary 4.2 when there are dense-order constraints. In doing so, we should evaluate P over an abstract interpretation of generalized tuples that are conjunctions of EDB literals and constraint literals; for full details, see [Levy93].

Finally, given an arbitrary datalog program P' (where EDB predicates are not necessarily unary), we can replace each EDB predicate of P' by the cartesian product of its columns, resulting in a program P. As earlier, we create the corresponding non-recursive program P_1. The relations computed by P_1 are upper bounds on those computed by P', provided that negated subgoals are ignored (since we cannot substitute an upper bound for a negated subgoal and still get an upper bound for the head). This could be useful when optimizing P', since P_1 is nonrecursive and, hence, some optimization problems are easier for P_1. For example, we may use P_1 (rather than P) for estimating sizes of IDB relations.

5 Undecidability for Monadic Recursion

In this section, we show that equivalence (and containment) is undecidable for datalog programs with unary IDB predicates, negation, and the interpreted predicate $\neq$. Since $\neq$ can be expressed in datalog with negation using a pair of nonrecursive binary predicates, the same result holds also for datalog programs with negation in which all recursive predicates are unary. The proof is based on a construction of a datalog program that checks whether a given computation of a two counter machine is a halting computation. The construction is similar to the one used in [GMSV87] to show that boundedness of datalog programs with unary IDB predicates and $\neq$ is undecidable.

The computation is stored in the EDB. The EDB predicate $cnfg(T, S, C_1, C_2)$ represents configurations of M, where T is the time, S is the state, and the values of the two counters are C_1 and C_2, respectively.

The states are represented by the integers 0, 1, ..., h, where 0 is the initial state and h is the halting (or accepting) state. The initial configuration is $cnfg(0, 0, 0, 0)$. In each move of the counter machine, the time is increased by 1 and the state and counters are changed according to the transition function of the machine.

The unary EDB predicate $zero(X)$ represents 0, and the binary EDB predicate $succ(X, Y)$ represents the successor relation. Note that all EDB relations are finite. In particular, *succ* represents only a finite portion of the successor relation, and *cnfg* represents only finitely many steps in a computation of M.

The EDB predicates *zero*, *succ*, and *cnfg* are assigned arbitrary relations. Therefore, there is no guarantee that they represent correctly what they are supposed to represent. For example, the relation for *zero* may contain several ground facts (and not just $zero(0)$), or it may contain none.

In the program we now write, there are rules for checking that the EDB predicates are correct. Each one of these rules expresses a condition that is satisfied if some aspect of the EDB is wrong, and if so, then the (0-arity) IDB predicate *bad* is assigned **true**.

Zero Check. The following rule assigns **true** to *bad* if there are two zeros.

$$bad \quad :- \quad zero(X),\ zero(Y),\ X \neq Y.$$

The next rule checks that zero is not the successor of another constant.

$$bad \quad :- \quad succ(X, Y),\ zero(Y).$$

Successor Check. The following rules check that no constant has either two successors or two predecessors.

$$bad \quad :- \quad succ(X, Y),\ succ(X, Z),\ Y \neq Z.$$
$$bad \quad :- \quad succ(Y, X),\ succ(Z, X),\ Y \neq Z.$$

The following rules check whether it is possible to count from 0 to h using *succ* (recall that the states of M are represented by 0, 1, ..., h, and we may assume that $h > 0$).

$$
\begin{aligned}
counth &:- zero(Z),\ succ(Z, V_1),\ succ(V_1, V_2),\ \ldots,\\
&\quad succ(V_{h-1}, S).\\
bad &:- \neg counth.
\end{aligned}
$$

The EDB relation for *succ* may have portions that are not reachable from zero. It is possible to write rules for checking that, but it is not really needed.

Initialization Check. The initial configuration is $cnfg(0,0,0,0)$. The following rules check that the configuration that has zero in the first argument position also has zero in every other argument position.

$$
\begin{aligned}
bad &:- cnfg(T, S, C_1, C_2),\ zero(T),\ S \neq T.\\
bad &:- cnfg(T, S, C_1, C_2),\ zero(T),\ C_1 \neq T.\\
bad &:- cnfg(T, S, C_1, C_2),\ zero(T),\ C_2 \neq T.
\end{aligned}
$$

Transition Check. Checking the transitions requires considering each pair of successive configurations and each transition. For example, consider the configurations $cnfg(T, S, C_1, C_2)$ and $cnfg(T', S', C_1', C_2')$, where T' is the successor of T, and the transition $\delta(j, >, =) = (j', pop, push)$. Note that this transition means that if the machine is in state j, the first counter is greater than zero, and the second counter is zero, then the machine moves to state j' while decrementing the first counter and incrementing the second counter. The following rule checks whether $cnfg(T, S, C_1, C_2)$ satisfies the condition of the transition (i.e., $(j, >, =)$) but the state in $cnfg(T', S', C_1', C_2')$ is not j'. There are similar rules for checking whether the counter C_1' is not the result of decrementing C_1 by one, or the counter C_2' is not the result of incrementing C_2 by one.

$$
\begin{aligned}
bad &:- cnfg(T, S, C_1, C_2),\ cnfg(T', S', C_1', C_2'),\\
&\quad succ(T, T'),\ S = j,\ succ(X, C_1),\ zero(C_2),\\
&\quad S'' = j',\ S' \neq S''.
\end{aligned}
$$

In the above rule, $S = j$ is a shorthand for the conjunction

$$zero(Z),\ succ(Z, V_1),\ succ(V_1, V_2),\ \ldots,\ succ(V_{j-1}, S)$$

(provided that j is not zero; otherwise, $S = j$ is just $zero(S)$). The atom $S'' = j'$ represents a similar conjunction.

Note that in the body of the above rule, the first line checks whether the two configurations are successive ones, the second line checks whether the first configuration satisfies the condition of the transition $\delta(j, >, =) = (j', pop, push)$, and the last line checks whether the state S' (of the second configuration) is not j'. Note that the atom $succ(X, C_1)$ checks that C_1 is greater than zero.

In summary, there are three rules of the above form for each transition: one rule for checking the state, and one rule for checking each counter. In all rules, the first line is the same as in the above rule; the second line depends on the condition for applying the transition (and is the same in all three rules for a given transition); the third line is different in each of the three rules for a given transition, depending on what is being checked—the state or one of the counters.

Computing Reachable Configurations. Even if the above rules do not assign **true** to *bad*, the EDB may have configurations that are not reachable from the initial one (or there may be no initial configuration).

The following rules compute the times of those configurations in the EDB that are reachable from the initial one, provided that no error is detected by the above rules.

$$
\begin{aligned}
reach(T) &:- cnfg(T, S, C_1, C_2),\ zero(T),\ \neg bad.\\
reach(T') &:- reach(T),\ cnfg(T', S', C_1', C_2'),\\
&\quad succ(T, T'),\ \neg bad.
\end{aligned}
$$

Halting Rule. The following rule assigns **true** to the (0-arity) IDB predicate *halt* if some configuration reachable from the initial one has the halting state and *bad* is not **true**.

$$halt :- reach(T),\ cnfg(T, S, C_1, C_2),\ S = h,\ \neg bad.$$

Proposition 5.1 *If bad is not assigned* **true**, *then exactly one connected component of succ is a correct representation of the integers from 0 to some n, and $n \geq h$ (where h is the largest state).* □

Proof: The EDB must have elements that corresponds to the integers from 0 to at least h (or else *counth* is not **true** and, hence, *bad* must be **true**). The rules for checking zero guarantee that there is no more than one zero in the EDB. Moreover, the connected component of *succ* that contains 0 does not have cycles, because 0 does not have any predecessor and every other element has at most one predecessor. Therefore, the connected component that starts at 0 is a correct representation of the integers from 0 to some n ($n \geq h$). □

Lemma 5.2 *If reach is not empty, then it contains all integers from 0 to some m, and for every i ($1 \leq i \leq m$), the correct configuration of M at time i is in the EDB.* □

Proof: Note that if *reach* is not empty, then it must contain 0 and the correct configuration at time 0 is in the EDB. Inductively, suppose that *reach* contains all the integers from 0 to $k-1$, and for each i ($1 \leq i \leq k-1$), the correct configuration of M at time i is in the EDB.

If either $k-1$ does not have a successor in the EDB or there is no configuration for time k in the EDB, then $\{0, 1, \ldots, k-1\}$ is the relation computed for *reach* and the claim is proved.

If the EDB has a successor of $k-1$ and a configuration for time k, then k is in the relation computed for *reach*. It remains to show that the configuration for time k is the correct one. Since *reach* is not empty, *bad* is not **true**, and so, the rules for checking transitions are not satisfied. However, this is not enough for showing that the transition is the correct one. Consider, for example, the rule that appears in the section on the transition check, and suppose that we substitute the configurations at time $k-1$ and time k for $cnfg(T, S, C_1, C_2)$ and $cnfg(T', S', C_1', C_2')$, respectively. If the configuration at time k is the wrong one, then *bad* becomes **true** only if the other atoms in the body are also satisfiable. The atoms $S = j$ and $S'' = j'$ are satisfiable, since the rule for *counth* (in the successor check) verifies the ability to count up to h (which is the largest state). The atom $succ(X, C_1)$ is satisfiable, since (by the inductive hypothesis) the configuration (substituted for) $cnfg(T, S, C_1, C_2)$ is the correct one for time $k-1$ and, so, the value substituted for C_1 is at most $k-1$ (because the time is incremented by one at each step, and therefore the counters cannot be larger than the time). For a similar reason, the atom $succ(C_1, X)$ is also satisfiable (since the counter is at most $k-1$ and we assume that $k-1$ has a successor). Note that the atom $succ(C_1, X)$ could be used in a rule that checks that at time k the first counter is the correct one. This proves the induction. □

Theorem 5.1 *Satisfiability of datalog programs with unary IDB predicates, negation, and the interpreted predicate $\neq$ is undecidable.*

Satisfiability is also undecidable for datalog programs that have unary IDB predicates, a pair of nonrecursive binary IDB predicates, and negation (but no interpreted predicates).

In both cases, negation is applied only to nonrecursive predicates, and there is a single recursive rule in each program. □

Proof: We construct a program P with all the rules mentioned above. The goal predicate of the program is *halt*.

Clearly, if the counter machine M reaches a halting state (when the computation starts with a 0 in each counter), then there is an EDB D, such that the evaluation of program P with respect to D assigns **true** to the IDB predicate *halt*.

Conversely, if program P assigns **true** to the predicate *halt* for some database D, then by Lemma 5.2, the configurations in the EDB describe a correct halting computation of M.

Therefore, program P is satisfiable (computes the value *true* for *halt* on some EDB D), if and only if the 2 counter machine M halts.

The above rules use the built-in predicate $\neq$. This predicate, however, can be expressed in nonrecursive datalog with negation and no interpreted predicates. In proof, the domain of the EDB (i.e., the set of all constants appearing in the EDB) can certainly be expressed in datalog. So, let the unary IDB predicate $dom(X)$ represent the domain. The following nonrecursive rules compute the predicate $\neq$.

$$\begin{aligned} equal(X, X) &:- dom(X),\ dom(X). \\ \neq(X, Y) &:- dom(X),\ dom(Y),\ \neg equal(X, Y). \end{aligned}$$

□

Remark 5.1 *The two rules involving counth (in the section on the successor check) are not really needed. Without these rules Lemma 5.2 may be incorrect. However, when $S = h$ cannot be satisfied (i.e., it is impossible to count from 0 to h using succ), predicate halt cannot be assigned* **true**. *Therefore, the above theorem remains correct even without those rules.*

6 Inferring Functional Dependencies

Solving the satisfiability problem for various languages also enables us to solve some inference problems for functional dependencies. Suppose a program P has an IDB predicate $p(U, V, X, Y, Z)$. The problem of *inferring* the functional dependency

$$FD(p, P) : \{1, 2\} \rightarrow \{3, 4\}$$

is deciding whether for all EDBs D, the relation for p computed by program P satisfies $\{1, 2\} \rightarrow \{3, 4\}$, i.e., if two tuples are equal on the first and second argument positions then they must also be equal on the third and fourth argument positions. Assuming that no constraints are imposed on the EDB, this problem can be reduced to a satisfiability problem as follows. Consider the following rules defining a new predicate q:

$$\begin{aligned} r_1: \quad q &:- p(U, V, X_1, Y_1, Z_1),\ p(U, V, X_2, Y_2, Z_2), \\ &\quad X_1 \neq X_2. \\ r_2: \quad q &:- p(U, V, X_1, Y_1, Z_1),\ p(U, V, X_2, Y_2, Z_2), \\ &\quad Y_1 \neq Y_2. \end{aligned}$$

Theorem 6.1 *If no constraints are imposed on the EDB, then predicate q is satisfiable in $P \cup \{r_1, r_2\}$ if and only if the functional dependency $FD(p, P) : \{1, 2\} \rightarrow \{3, 4\}$ cannot be inferred.* □

Datalog Extension/Restriction	Sat/QR	EQ/CN
Unary idb's	*Decidable*	Decidable [CGKV88]
Unary idb's, negation on nonrecursive predicates, 1 recursive rule, $\neq$	**Undecidable**	*Undecidable*
Unary recursive idb's, binary nonrecursive idbs, negation on nonrecursive predicates, 1 recursive rule, no $\neq$	**Undecidable**	*Undecidable*
Binary idb's	Decidable [Shm87]	Undecidable [Shm87, UV88]
Dense order constraints	Decidable [KKR90, LS92]	Undecidable
Unary edb's, stratified negation	*Decidable*	**Decidable**
Dense order constraints, negation on edbs	**Decidable**	Undecidable
$\neq$, functional dependencies in edb	**Undecidable**	*Undecidable*

Table 1: The Decidability Question

Corollary 6.2 *Satisfiability is undecidable for datalog programs with $\neq$, but without negation or any other interpreted predicates, if EDBs are required to satisfy some functional dependencies.* □

Proof: Follows from the above reduction and the proof of [AH88] that inference of functional dependencies in datalog programs is undecidable. □

7 Conclusions

We identified new classes of programs for which equivalence, query-reachability and satisfiability are either decidable or undecidable. Table 1 summarizes the classes of programs for which answers to the decidability problems are now available. We have used bold font to indicate the contributions of this paper. Previously known results are in regular font. Italic font indicates results that follow from previous and new results by applying the Lemmas and Propositions of Section 2. Results that follow trivially from previously known results are shown in small teletype font.

In [vdM92], van der Meyden proved an undecidability result that implies the following. First, satisfiability is undecidable for datalog programs with unary IDB predicates, $\neq$ and negation. Second, containment of a recursive program in a nonrecursive program is undecidable even if there are only unary IDB predicates and $\neq$ (i.e., negation is not needed).

The first result of [vdM92] is similar to our result in Theorem 5.1 on undecidability of satisfiability. As for the second result of [vdM92], we can modify the proof in Section 5 to show the same (details will be given in the full paper). It should be noted, however, that the proof in [vdM92] uses many recursive rules, compared to just one recursive rule in our proof.

Undecidability of the containment of a monadic datalog program, with a single linear recursive rule, in a monadic nonrecursive program with $\neq$ should be contrasted with the following results: First, containment of a recursive datalog program in a nonrecursive program, where IDB predicates are of any arity but there is neither negation nor $\neq$, is decidable [CV92]. Second, containment between two recursive datalog programs with only unary predicates (and neither negation nor $\neq$) is decidable [CGKV88].

Acknowledgment

The authors thank Ron van der Meyden for pointing out the connection between his work and this paper.

References

[AH88] Serge Abiteboul and Richard Hull. Data functions, datalog, and negation. In *Proceedings of ACM SIGMOD 1988 International Conference on Management of Data*, Chicago, IL, June 1988.

[CV92] Surajit Chaudhuri and Moshe Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, pages 55–66, San Diego, CA, June 2-4 1992. ACM SIGACT-SIGMOD-SIGART.

[CGKV88] S. S. Cosmadakis, H. Gaifman, Paris C. Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic

programs. In *Proceedings of the Twentieth Symposium on Theory of Computing*, pages 477–490, 1988.

[GMSV87] H. Gaifman, H. Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science(LICS)*, pages 106–115, Ithaca, NY, 1987.

[KKR90] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. In *Proceedings of the Ninth Symposium on Principles of Database Systems (PODS)*, pages 299–313, Nashville, TN, April 2-4 1990. ACM SIGACT-SIGMOD-SIGART.

[Levy93] Alon Y. Levy. Relevance Reasoning in Knowledge Based Systems. *Ph.D Thesis, Stanford University, 1993.*

[LS92] Alon Levy and Yehoshua Sagiv. Constraints and redundancy in datalog. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, pages 67–80, San Diego, CA, June 2-4 1992. ACM SIGACT-SIGMOD-SIGART.

[Shm87] Oded Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the Sixth Symposium on Principles of Database Systems (PODS)*, pages 237–249, San Diego, CA, March 1987. ACM SIGACT-SIGMOD-SIGART.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1.* Computer Science Press, 1988.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 2.* Computer Science Press, 1989.

[UV88] Jeffrey D. Ullman and Allen Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, **3**:5–42, 1988.

[vdM92] Ronald van der Meyden. *The Complexity of Querying Indefinite Information: Defined Relations, Recursion and Linear Order.* PhD thesis, Rutgers, The State University of New Jersey, New Brunswick, NJ, October 1992.

[Var89] Moshe Y. Vardi. Automata theory for database theoreticians. In *Proceedings of the Eighth Symposium on Principles of Database Systems (PODS)*, pages 83–92, Philadelphia, PA, March 1989. ACM SIGACT-SIGMOD-SIGART.