Queries Independent of Updates

Alon Y. Levy*
Department of Computer Science
Stanford University
Stanford, California, 94305
levy@cs.stanford.edu

Yehoshua Sagiv†
Department of Computer Science
Hebrew University
Jerusalem, Israel
sagiv@cs.huji.ac.il

Abstract

This paper considers the problem of detecting independence of a queries expressed by datalog programs from updates. We provide new insight into the independence problem by reducing it to the equivalence problem for datalog programs (both for the case of insertion and deletion updates). Equivalence, as well as independence, is undecidable in general. However, algorithms for detecting subclasses of equivalence provide sufficient (and sometimes also necessary) conditions for independence. We consider two such subclasses. The first, *query-reachability*, generalizes previous work on independence [BCL89, El90], which dealt with nonrecursive programs with a single occurrence of the updated predicate. Using recent results on query-reachability [LS92, LMSS93], we generalize these earlier independence tests to arbitrary recursive datalog queries with dense-order constraints and negated EDB subgoals. The second subclass is *uniform equivalence* (introduced in [Sa88]). We extend the results of [Sa88] to datalog programs that include dense-order constraints and stratified negation. Based on these extensions, we present new cases in which independence is decidable and give algorithms that are sound for the general case. Aside for their use in detecting independence, the algorithms for detecting uniform equivalence are also important for optimizing datalog programs.

*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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993.

1 Introduction

We consider the problem of detecting independence of queries expressed by datalog programs from updates. Detecting independence is important for several reasons. It can be used in view maintenance to identify that some views are independent of certain updates. In transaction scheduling, we can provide greater flexibility by identifying that one transaction is independent of updates made by another. Finally, we can use independence in query optimization by ignoring parts of the database for which updates do not affect a specific query.

In this paper, we provide new insight into the independence problem by reducing it to the equivalence problem for datalog programs. Equivalence, as well as independence, is undecidable in general. However, algorithms for equivalence provide sufficient (and sometimes also necessary) conditions for independence. We consider two such conditions, *query reachability* [LS92] and *uniform equivalence* [Sa88].

Earlier work by Blakeley et al. [BCL89] and Elkan [El90] focussed on cases for which independence is the same as query reachability. Essentially, these are the cases where the updated predicate has a single occurrence in the query. Blakeley et al. [BCL89] considered only conjunctive queries. Elkan [El90] considered a more general framework, but gave an algorithm only for nonrecursive rules without negation; that algorithm is complete only for the case of a single occurrence of the updated predicate. Elkan also gave a proof method for recursive rules, but its power is limited.

Query reachability has recently been shown decidable even for recursive datalog programs with dense-order constraints and negated EDB subgoals [LS92, LMSS93]. We show how query-reachability algorithms generalize the previous results on independence.

In order to use uniform equivalence for detecting independence, we extend the algorithm given in [Sa88] to datalog programs with built-in predicates and

stratified negation. As a result, we show new decidable cases of independence; for example, if the update is an insertion, and both the query and the update are given by datalog program with no recursion or negation, then independence is decidable (note that the updated predicate may have multiple occurrences). Our algorithms also provide sufficient conditions for independence in the general case. Aside from their usage in detecting independence, the algorithms we present for uniform equivalence are important for optimizing datalog programs.

Finally, we also characterize new cases for which independence of insertions is the same as independence of deletions. Since the former is, in many cases, easier to detect, these characterizations are of practical importance.

2 Preliminaries

2.1 Datalog Programs

Datalog programs are collections of safe horn-rules with no function symbols (i.e., only constants and variables are allowed). We allow the built-in predicates[1] $<$, $>$, $=$, $\neq$, $\leq$, and $\geq$ that represent a dense order. Programs may also have stratified negation. Both negation and built-in predicates must be used safely (cf. [Ull88]).

We distinguish between two sets of predicates in a given program: the *extensional predicates* (EDB predicates), which are those that appear only in bodies of rules, and the *intensional predicates* (IDB predicates), which are the predicates appearing in heads of rules. The EDB predicates refer to the database relations while the IDB predicates are defined by the program. We usually denote the EDB predicates as $e_1, \ldots, e_m$ and the IDB predicates as $i_1, \ldots, i_n$. The input to a datalog program is an *extensional database* (EDB) consisting of relations $E_1, \ldots, E_m$ for the EDB predicates $e_1, \ldots, e_m$, respectively. Alternatively, the EDB may also be viewed as a set of ground atoms (or facts) for the EDB predicates. Given a datalog program $\mathcal{P}$ and an EDB $E_1, \ldots, E_m$ as input, a bottom-up evaluation is one in which we start with the ground EDB facts and apply the rules to derive facts for the IDB predicates. We continue applying the rules until no new facts are generated. We distinguish one IDB predicate as the *query* (or goal) predicate, and the *output* (or answer) of program $\mathcal{P}$ for the input $E_1, \ldots, E_m$, denoted $\mathcal{P}(E_1, \ldots, E_m)$, is the set of all ground facts generated for the query predicate in the bottom-up evaluation. The query predicate is usually denoted as q. Note that the bottom-up evaluation computes relations for all the IDB predicates, and $I_1, \ldots, I_n$ usually denote the relations for the IDB predicates $i_1, \ldots, i_n$, respectively.

[1] The phrase "built-in predicates" refers in this paper just to those listed above.

We say that the query predicate is monotonic (anti-monotonic) in the input if whenever $D_1 \supseteq D_2$ then $P(D_1) \supseteq P(D_2)$ ($P(D_1) \subseteq P(D_2)$). Note that a datalog program without negation is monotonic.

2.2 Containment and Equivalence

Independence of queries from updates can be expressed as an equivalence of two programs: one program that computes the answer to the query before the update and a second program that computes the answer after the update.

Definition 2.1: **(Containment)** A datalog program $\mathcal{P}_1$ *contains* a program $\mathcal{P}_2$, written $\mathcal{P}_2 \subseteq \mathcal{P}_1$, if for all EDBs $E_1, \ldots, E_m$, the output of $\mathcal{P}_1$ contains that of $\mathcal{P}_2$, i.e., $\mathcal{P}_2(E_1, \ldots, E_m) \subseteq \mathcal{P}_1(E_1, \ldots, E_m)$. ∎

Two programs $\mathcal{P}_1$ and $\mathcal{P}_2$ are *equivalent*, written $\mathcal{P}_1 \equiv \mathcal{P}_2$, if $\mathcal{P}_2 \subseteq \mathcal{P}_1$ and $\mathcal{P}_1 \subseteq \mathcal{P}_2$. Containment of datalog programs is undecidable [Sh87], even for programs without built-in predicates or negation.

A sufficient condition for containment is *uniform containment*, which was introduced and shown to be decidable in [Sa88] for programs without built-in predicates or negation. In defining uniform containment, we assume that the input to a program $\mathcal{P}$ consists of EDB relation $E_1, \ldots, E_m$ as well as *initial* IDB relations $I_1^0, \ldots, I_n^0$ for the IDB predicates. The output of program $\mathcal{P}$ for $E_1, \ldots, E_m, I_1^0, \ldots, I_n^0$, written $\mathcal{P}(E_1, \ldots, E_m, I_1^0, \ldots, I_n^0)$, is computed as earlier by applying rules bottom-up until no new facts are generated. When dealing with uniform containment (equivalence), we assume that the output is not just the relation for the query predicate but rather all the IDB relations $I_1, \ldots, I_n$ computed for the IDB predicates $i_1, \ldots, i_n$, respectively. An output $I_1, \ldots, I_n$ *contains* another output $I'_1, \ldots, I'_n$ if $I'_j \subseteq I_j$ ($1 \leq j \leq n$). A program $\mathcal{P}_1$ *uniformly contains* $\mathcal{P}_2$, written $\mathcal{P}_2 \subseteq^u \mathcal{P}_1$, if for all EDBs $E_1, \ldots, E_m$ and for all initial IDBs $I_1^0, \ldots, I_n^0$,

$$\mathcal{P}_2(E_1, \ldots, E_m, I_1^0, \ldots, I_n^0) \subseteq \mathcal{P}_1(E_1, \ldots, E_m, I_1^0, \ldots, I_n^0).$$

Uniform containment can also be explained in model-theoretic terms [Sa88]. For programs without negations, the uniform containment $\mathcal{P}_2 \subseteq^u \mathcal{P}_1$ holds if and only if $M(\mathcal{P}_1) \subseteq M(\mathcal{P}_2)$, where $M(\mathcal{P}_i)$ denotes the set of all models of $\mathcal{P}_i$. Furthermore, for programs having only EDB predicates in bodies of rules, uniform containment is the same as containment. Note that a program with no recursion (and no negation) can be transformed into this form by unfolding the rules.

Query reachability is another notion that provides a sufficient condition for equivalence of programs.

Definition 2.2: (Query Reachability) Let p be a predicate (either EDB or IDB) of a program $\mathcal{P}$. The predicate p is *query reachable* with respect to $\mathcal{P}$ if there is a derivation d of a fact for the query predicate from some EDB D, such that predicate p is used in d. ∎

Algorithms for deciding query reachability are discussed in [LS92, LMSS93] for cases that include built-in predicates and negation.

2.3 Updates

Given a Datalog program $\mathcal{P}$, which we call the *query program,* we consider updates to the EDB predicates of $\mathcal{P}$, denoted $e_1, \ldots, e_m$. In an update, we either remove or add ground facts to the extensional database. To simplify notation, we assume that updates are always done on the relation E_1 for the predicate e_1. To specify the set of facts that is updated in E_1, we assume we have another datalog program, called the *update program* and denoted as $\mathcal{P}^u$. The query predicate of $\mathcal{P}^u$ is u and its arity is equal to that of e_1. The relation computed for u will be the set of facts updated in e_1.

We assume that the IDB predicates of $\mathcal{P}^u$ are different from those of $\mathcal{P}$. The EDB predicates of $\mathcal{P}^u$, however, could be EDB predicates of $\mathcal{P}$ as well as predicates not appearing in $\mathcal{P}$. To distinguish the two sets of EDB predicates, from now on the phrase "EDB predicates" refers exclusively to the EDB predicates $e_1, \ldots, e_m$ of the query program $\mathcal{P}$; the other extensional relations that may appear in the update program are referred to as *base predicates,* denoted by $b_1, \ldots, b_l$. We denote the output of the update program $\mathcal{P}^u$ as $\mathcal{P}^u(E_1, \ldots, E_m, B_1, \ldots, B_l)$, even if $\mathcal{P}^u$ does not use all (or any) of the EDB predicates. Sometimes we refer to its output as U.

An update is either an *insertion* or a *deletion* and it applies to the relation E_1 for the EDB predicate e_1. The tuples to be inserted into or deleted from E_1 are those in the relation computed for u. A large class of updates consists of those *not* depending on the EDB relations, as captured by the following definition:

Definition 2.3: (Oblivious Update) An update specified by an update program $\mathcal{P}^u$ is *oblivious* with respect to a query program $\mathcal{P}$ if $\mathcal{P}^u$ has only base predicates (and no EDB predicates). An update is *nonoblivious* if the update program $\mathcal{P}^u$ has some EDB predicates (and possibly some base predicates). ∎

To define independence, suppose we are given a query program $\mathcal{P}$ and an update program $\mathcal{P}^u$. The program $\mathcal{P}$ is *independent* of the given update if the update does not change the answer to the query predicate. More precisely, program $\mathcal{P}$ is independent of the given update if for all EDB relations $E_1, \ldots, E_m$ and for all base relations $B_1, \ldots, B_l$,

$$\mathcal{P}(E_1, E_2, \ldots, E_n) = \mathcal{P}(E_1', E_2, \ldots, E_n)$$

where E_1' is the result of applying the update to E_1; that is, $E_1' = E_1 \cup U$ if the update is an insertion and $E_1' = E_1 - U$ if the update is a deletion, where $U = \mathcal{P}^u(E_1, \ldots, E_m, B_1, \ldots, B_l)$.

We use the following notation. $In^+(\mathcal{P}, \mathcal{P}^u)$ means that program $\mathcal{P}$ is independent of the insertion specified by the update program $\mathcal{P}^u$. Similarly, $In^-(\mathcal{P}, \mathcal{P}^u)$ means that program $\mathcal{P}$ is independent of the deletion specified by the update program $\mathcal{P}^u$.

Several properties of independence are shown by Elkan [El90]. In particular, he showed the following.

Lemma 2.4: *Consider a query program $\mathcal{P}$ and an update program $\mathcal{P}^u$. If $\mathcal{P}^u$ is monotonic in the EDB predicates and $\mathcal{P}$ is either monotonic or anti-monotonic in the EDB predicates, then*

$$In^-(\mathcal{P}, \mathcal{P}^u) \Longrightarrow In^+(\mathcal{P}, \mathcal{P}^u).$$

Similarly to the above lemma, we can also prove the following.

Lemma 2.5: *Consider a query program $\mathcal{P}$ and an update program $\mathcal{P}^u$. If $\mathcal{P}^u$ is anti-monotonic in the EDB predicates and $\mathcal{P}$ is either monotonic or anti-monotonic in the EDB predicates, then*

$$In^+(\mathcal{P}, \mathcal{P}^u) \Longrightarrow In^-(\mathcal{P}, \mathcal{P}^u).$$

Proof: Consider an EDB $E_1, \ldots, E_n$, denoted as $\bar{E}$, and relations $B_1, \ldots, B_m$, denoted as $\bar{B}$, for the base predicates. The tuples of the update are given by $U = \mathcal{P}^u(\bar{E}, \bar{B})$. A deletion update transforms the EDB $\bar{E}$ into the EDB $E_1 - U, \ldots, E_n$, denoted as $\bar{E}^-$. We have to show the following.

$$\mathcal{P}(\bar{E}^-) \quad = \quad \mathcal{P}(\bar{E})$$

So, consider the EDB $\bar{E}^-$ with the relations $\bar{B}$ for the base predicates. Let $U' = \mathcal{P}^u(\bar{E}^-, \bar{B})$. Since $\mathcal{P}^u$ is anti-monotonic in the EDB, $U \subseteq U'$.

We now apply the insertion update specified by $U' = \mathcal{P}^u(\bar{E}^-, \bar{B})$ to $\bar{E}^-$ yielding the following EDB.

$$(E_1 - U) \cup U', E_2, \ldots, E_n$$

Since $In^+(\mathcal{P}, \mathcal{P}^u)$ is assumed, we get the following.

$$\mathcal{P}(\bar{E}^-) \quad = \quad \mathcal{P}((E_1 - U) \cup U', E_2, \ldots, E_n) \quad (1)$$

Moreover, $U \subseteq U'$ implies the following.

$$E_1 - U \subseteq E_1 \subseteq (E_1 - U) \cup U' \qquad (2)$$

If $\mathcal{P}$ is monotonic in the EDB, then (2) implies

$$\mathcal{P}(\bar{E}^-) \subseteq \mathcal{P}(\bar{E}) \subseteq \mathcal{P}((E_1 - U) \cup U', E_2, \ldots, E_n)$$

and, so, from (1) we get the following.

$$\mathcal{P}(\bar{E}^-) = \mathcal{P}(\bar{E})$$

Similarly, if $\mathcal{P}$ is anti-monotonic in the EDB, then (2) implies

$$\mathcal{P}(\bar{E}^-) \supseteq \mathcal{P}(\bar{E}) \supseteq \mathcal{P}((E_1 - U) \cup U', E_2, \ldots, E_n)$$

and, so, from (1) we get the following.

$$\mathcal{P}(\bar{E}^-) = \mathcal{P}(\bar{E})$$

Note that if an update is oblivious, then it is both monotonic and anti-monotonic. Therefore, the above two lemmas imply the following corollary.

Corollary 2.6: *Consider a query program $\mathcal{P}$ and an update program $\mathcal{P}^u$. If the update is oblivious (i.e., EDB predicates of $\mathcal{P}$ do not appear in $\mathcal{P}^u$), and $\mathcal{P}$ is either monotonic or anti-monotonic in the updated EDB predicates, then the following equivalence holds:*

$$In^-(\mathcal{P}, \mathcal{P}^u) \Longleftrightarrow In^+(\mathcal{P}, \mathcal{P}^u).$$

The importance of Lemma 2.5 and Corollary 2.6, as we will see in the next section, lies in the fact that testing $In^+(\mathcal{P}, \mathcal{P}^u)$ is usually easier than testing $In^-(\mathcal{P}, \mathcal{P}^u)$.

3 Detecting Independence

To develop algorithms for detecting independence, we will show that the problem can be reformulated as a problem of detecting equivalence of datalog programs. Like independence, detecting equivalence of datalog programs is in general undecidable; however, algorithms that provide sufficient conditions for detecting equivalence can also serve as sufficient conditions for independence. In contrast, previous work reduced the independence problem to satisfiability. The following example illustrates the difference between the approaches.

Example 3.1: Consider the following program P_1. An atom $canDrive(X, Y, A)$ is true if person X can drive car Y and A is the age of X. According to the rule for $canDrive$, person X can drive car Y if X is a driver and there is someone of the age 18 or older in the same car. An adult driver, as computed by the IDB predicate $adultDriver$, is anyone who can drive a car and is of the age 18 or older.

$$\begin{array}{rcl} canDrive(X,Y,A) & :- & inCar(X,Y,A),\ driver(X), \\ & & inCar(Z,Y,B),\ B \geq 18. \\ adultDriver(X) & :- & canDrive(X,Y,A),\ A \geq 18. \end{array}$$

Let the update program consist of the rule

$$\begin{array}{rcl} u_1(X,Y,A) & :- & inCar(X,Y,A),\ \neg driver(X), \\ & & A < 18. \end{array}$$

and suppose that the deletion defined by u_1 is applied to $inCar$; that is, non-drivers under the age of 18 are removed from $inCar$.

Let the query predicate be $adultDriver$ and note that $adultDriver(X)$ is equivalent to the following conjunction, denoted as $\mathcal{C}$.

$$inCar(X,Y,A) \wedge driver(X) \wedge inCar(Z,Y,B) \wedge A \geq 18 \wedge B \geq 18$$

An algorithm for detecting independence based on satisfiability (e.g., [El90, BCL89]) checks whether an updated fact may appear in any derivation of the query. In our example, an updated fact may appear in a derivation of $adultDriver(X)$ if either the conjunction

$$\mathcal{C} \wedge \neg driver(X) \wedge A < 18$$

or the conjunction

$$\mathcal{C} \wedge \neg driver(Z) \wedge B < 18$$

is satisfiable. Since none of the above is satisfiable, the algorithm would conclude that the query is independent of the update.

Now consider the following update program

$$u_2(X,Y,A) \ :- \ inCar(X,Y,A),\ \neg driver(X).$$

and suppose that the deletion defined by u_2 is applied to $inCar$. In this case, the conjunction

$$\mathcal{C} \wedge \neg driver(Z)$$

is satisfiable and the algorithm would *not* detect independence. However, to see that the update is independent, observe that *after* the update, P_1 computes for $adultDriver$ the same relation as the one computed by the following program, P_2, *before* the update.

$$\begin{array}{rcl} canDrive(X,Y,A) & :- & inCar(X,Y,A),\ driver(X), \\ & & inCar(Z,Y,B),\ B \geq 18, \\ & & driver(Z). \\ adultDriver(X) & :- & canDrive(X,Y,A),\ A \geq 18. \end{array}$$

Since P_1 and P_2 are equivalent (when the query predicate is $adultDriver$), the deletion update defined by u_2 is independent of the query predicate. ∎

3.1 Independence and Equivalence

As stated above, the independence problem can be formulated as a problem of detecting equivalence of datalog programs. To show that, we construct a new program that computes the new value of the query predicate q from the old value of the EDB (i.e., the value before the update). One program, $\mathcal{P}^+$, is constructed for the case of insertion, and another program, $\mathcal{P}^-$, is constructed for the case of deletion. Each of $\mathcal{P}^+$ and $\mathcal{P}^-$ consists of three parts:

- The rules of $\mathcal{P}$, after all occurrences of the predicate name e_1 have been replaced by a new predicate name s.
- The rules of the update program $\mathcal{P}^u$.
- Rules for the new predicate s.

$\mathcal{P}^+$ and $\mathcal{P}^-$ differ only in the third part. In the case of insertion, the predicate s in $\mathcal{P}^+$ is intended to represent the relation E_1 after the update, and therefore the rules for s are:

$$s(X_1,\ldots,X_k) \;:\!-\; e_1(X_1,\ldots,X_k).$$
$$s(X_1,\ldots,X_k) \;:\!-\; u(X_1,\ldots,X_k).$$

In the case of deletion, the predicate s in $\mathcal{P}^-$ is intended to represent the deletion update to E_1, and the rule for defining it is

$$s(X_1,\ldots,X_k) \;:\!-\; e_1(X_1,\ldots,X_k),\ \neg u(X_1,\ldots,X_k).$$

The following propositions are immediate corollaries of the definition of independence.

Proposition 3.2: $In^+(\mathcal{P},\mathcal{P}^u) \Longleftrightarrow \mathcal{P} \equiv \mathcal{P}^+$.

Proposition 3.3: $In^-(\mathcal{P},\mathcal{P}^u) \Longleftrightarrow \mathcal{P} \equiv \mathcal{P}^-$.

Proof: Both propositions follow from the observation that the relation computed for s is the updated relation for E_1. Therefore, since e_1 is replaced by s in the rules of the program, the new program will compute the relation for q after the update. Clearly, the independence holds if and only if the new program is equivalent to the original program. ∎

Example 3.4: Consider the following program $\mathcal{P}_0$ with q as the query predicate:

$$r_1: \; q(X,Y) \;:\!-\; p(X,Y),\ \neg e_1(X,Y).$$
$$r_2: \; p(X,Y) \;:\!-\; e_1(X,Y),\ \neg e(X).$$
$$r_3: \; p(X,Y) \;:\!-\; e_1(X,Y),\ X > 1.$$
$$r_4: \; p(X,Y) \;:\!-\; e(X),\ e(W),\ p(W,Y),\ W > X.$$

Let the update program $\mathcal{P}^u$ consist of the rule:

$$r^u: \; u(X,Y) \;:\!-\; b(X,Y),\ X \le 1.$$

The program for the insertion update $\mathcal{P}^+$ would be:

$$r'_1: \; q(X,Y) \;:\!-\; p(X,Y),\ \neg s(X,Y).$$
$$r'_2: \; p(X,Y) \;:\!-\; s(X,Y),\ \neg e(X).$$
$$r'_3: \; p(X,Y) \;:\!-\; s(X,Y),\ X > 1.$$
$$r'_4: \; p(X,Y) \;:\!-\; e(X),\ e(W),\ p(W,Y),\ W > X.$$
$$r'_5: \; s(X,Y) \;:\!-\; e_1(X,Y).$$
$$r'_6: \; s(X,Y) \;:\!-\; u(X,Y).$$
$$r'_7: \; u(X,Y) \;:\!-\; b(X,Y),\ X \le 1.$$

This program is equivalent to the original one, $\mathcal{P}_0$, and indeed $In^+(\mathcal{P}_0,\mathcal{P}^u)$ does hold. ∎

In Section 4, we describe algorithms for deciding uniform equivalence for datalog programs with built-in predicates and stratified negation. Based on these algorithms, we get the following decidability results for independence. Note that in the following theorem, the updated predicate may have multiple occurrences, and so, this theorem generalizes earlier results on decidability of independence.

Theorem 3.5: *Independence is decidable in the following cases:*

1. *$In^+(\mathcal{P},\mathcal{P}^u)$ ($In^-(\mathcal{P},\mathcal{P}^u)$) is decidable if both $\mathcal{P}^+$ ($\mathcal{P}^-$) and $\mathcal{P}$ have only built-in and EDB predicates (that may appear positively or negatively) in bodies of rules.*[2]
2. *Both $In^+(\mathcal{P},\mathcal{P}^u)$ and $In^-(\mathcal{P},\mathcal{P}^u)$ are decidable if $\mathcal{P}$ has only built-in and EDB predicates (that may appear positively or negatively) in bodies of rules, and $\mathcal{P}^u$ has only rules of the form*

   $$u(X_1,\ldots,X_k) \;:\!-\; e_1(X_1,\ldots,X_k),\ c.$$

   where c is a conjunction of built-in predicates.

The theorem follows from the observation that for these classes of programs, uniform equivalence is also a necessary condition for equivalence. The algorithms of Section 4 also apply to arbitrary programs $\mathcal{P}$, $\mathcal{P}^+$ and $\mathcal{P}^-$, but only as a sufficient condition for independence.

3.2 Independence and Satisfiability

Detecting independence based on satisfiability is based on the observation that if none of the updated facts can be part of a derivation of the query, then clearly, the query is independent of the update. This is made precise by the following lemma, based on query reachability.

[2] We prefer to describe this case in terms of $\mathcal{P}$ and $\mathcal{P}^+$, rather than $\mathcal{P}$ and $\mathcal{P}^u$, since it is clearer. Note that if $\mathcal{P}$ and $\mathcal{P}^u$ are nonrecursive then in some, but not all cases, $\mathcal{P}^+$ and $\mathcal{P}$ can be converted by unfolding into forms satisfying this condition.

Lemma 3.6: *Suppose that neither $\mathcal{P}$ nor $\mathcal{P}^u$ has negation. If predicate u is not query reachable in $\mathcal{P}^+$, then both $In^+(\mathcal{P},\mathcal{P}^u)$ and $In^-(\mathcal{P},\mathcal{P}^u)$ are true.*

Query reachability is decidable for all datalog programs with built-in predicates and negation applied to EDB (and base) predicates [LS92, LMSS93]. If negation is also applied to IDB predicates, then a generalization of the algorithm of [LMSS93] is a sufficient test for query reachability. Thus, the above lemma provides a considerable generalization of previous algorithms for detecting independence.

It should be realized that the independence tests of Elkan [El90] and of Blakely et al. [BCL89] are just query reachability tests. Both essentially characterized special cases in which independence is equivalent to query reachability. The result of Blakely et al. [BCL89] applies just to conjunctive queries with no repeated predicates. The work of Elkan [El90] entails that, in the case of recursive rules, independence is equivalent to query reachability provided that the updated predicate has a single occurrence; he also required that an insertion update be monotonic. For testing independence, Elkan [El90] gave a query-reachability algorithm for the case of nonrecursive, negation-free rules, and suggested a proof method for the recursive case; there is no characterization of the power of that proof method, but it should be noted that it cannot capture all cases detected by the algorithms of [LS92, LMSS93].

Example 3.7: The following example shows how query reachability can be used for detecting independence in the case of a recursive datalog program. Consider the following rules:

$$\begin{array}{lrcl} r_1: & goodPath(X,Y) & :- & badPoint(X), \\ & & & path(X,Y), \\ & & & goodPoint(Y). \\ r_2: & path(X,Y) & :- & link(X,Y). \\ r_3: & path(X,Y) & :- & link(X,Z),\ path(Z,Y). \\ r_4: & link(X,Y) & :- & step(X,Y). \\ r_5: & link(X,Y) & :- & bigStep(X,Y). \end{array}$$

The predicates *step* and *bigStep* describe single links between points in a space. The predicate *path* denotes the paths that can be constructed by composing single links. The predicate *goodPath* denotes paths that go from bad points to good ones. Furthermore, the following constraint are given on the EDB relations:
$badPoint(x) \Rightarrow 100 < x < 200.$
$step(x,y) \Rightarrow x < y.$
$goodPoint(x) \Rightarrow 150 < x < 170.$
$bigStep(x,y) \Rightarrow x < 100 \wedge y > 200.$

Figure 1 show the query-tree representing all possible derivation of the query $goodPath(X,Y)$. The query-tree shows that ground facts of the relation *step* which do not satisfy $100 < x$ and $y < 170$ cannot be part of a derivation of the query. Similarly, facts of the relation *bigStep* cannot be part of derivations of the query. Consequently, the query *goodPath* will be independent of removing or adding facts of that form. ∎

4 Uniform Equivalence

In this section, we describe algorithms for deciding uniform equivalence of programs that have built-in predicates and stratified negation. This extends a previous algorithm [Sa88] that dealt with datalog programs without built-in predicates or negations.

As shown in [Sa88], uniform containment (and equivalence) can be given model-theoretic characterization, namely, the uniform containment $\mathcal{P}_2 \subseteq^u \mathcal{P}_1$ holds if and only if $M(\mathcal{P}_1) \subseteq M(\mathcal{P}_2)$, where $M(\mathcal{P}_i)$ denotes the set of all models of $\mathcal{P}_i$. We note that $M(\mathcal{P}_1) \subseteq M(\mathcal{P}_2)$ holds if and only if $M(\mathcal{P}_1) \subseteq M(r)$ for every rule $r \in \mathcal{P}_2$, since a database D is a model of P_2 if and only if it is a model of every rule $r \in \mathcal{P}_2$. Our algorithms will decide whether $M(\mathcal{P}_1) \subseteq M(\mathcal{P}_2)$ by checking whether $M(\mathcal{P}_1) \subseteq M(r)$ for every $r \in \mathcal{P}_2$. We first discuss programs with only built-in predicates.

4.1 Uniform Containment with Built-in Predicates

When the programs have no interpreted predicates, the following algorithm (from [Sa88]) will decide whether a given rule r is uniformly contained in a program $\mathcal{P}$. Given a rule r of the form

$$p \ :- \ q_1, \ \ldots, \ q_n.$$

where p is the head of the rule and $q_1,\ldots,q_n$ are its subgoals, we use a substitution θ that maps every variable in the body of r to a distinct symbol that does not appear in $\mathcal{P}$ or r. We then apply the program $\mathcal{P}$ to the atoms $q_1\theta,\ldots,q_n\theta$. In [Sa88] it is shown that the program $\mathcal{P}$ generates $p\theta$ from $q_1\theta,\ldots,q_n\theta$ if and only if $M(\mathcal{P}) \subseteq M(r)$.

However, there is a problem in applying this algorithm to programs with interpreted predicates. First, the constants used in the input to $\mathcal{P}$, i.e., those that appear in $q_1\theta,\ldots,q_n\theta$, are arbitrary, and therefore, order relations are not defined on them. Consequently, the interpreted subgoals in the rules (that may involve $<, \leq$, etc.) can not be evaluated. Moreover, some of the derivations of $p\theta$ by $\mathcal{P}$ depend on the symbols satisfying the interpreted constraints, and so these cannot be discarded.

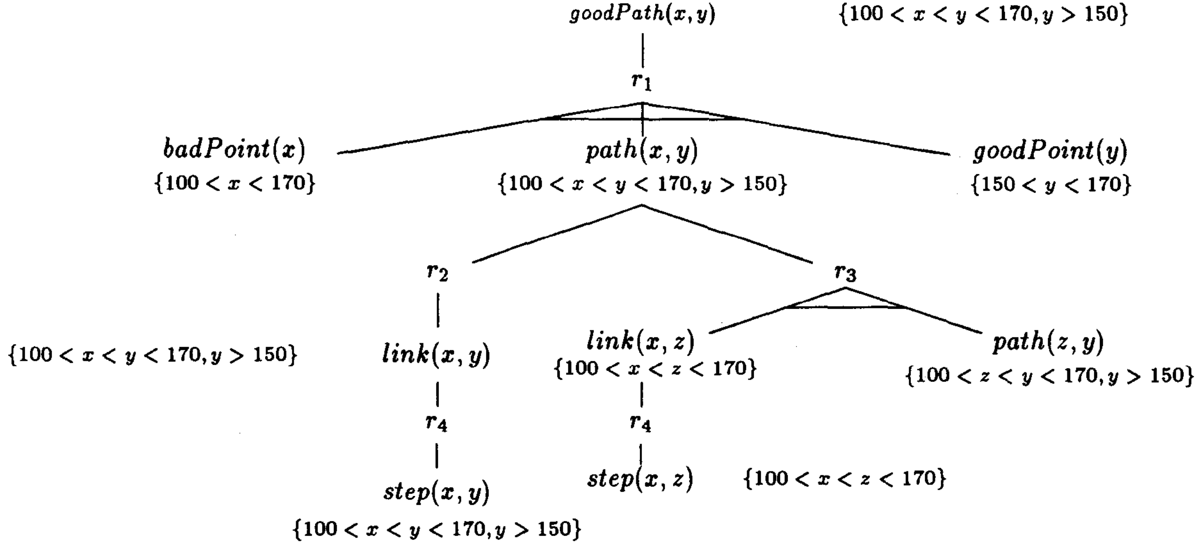

Figure 1: Detecting independence using query reachability

We address this problem by associating a constraint with every fact involved in the evaluation of $\mathcal{P}$. The constraints for a given fact f represent the conditions on $q_1\theta, \ldots, q_n\theta$ under which f is derivable. We manipulate these constraints as we evaluate $\mathcal{P}$. Formally, let r be the rule:

$$p \ :- \ q_1, \ \ldots, \ q_n, \ c_r. \tag{3}$$

We denote the set of variables in r by Y. The subgoal c_r is the conjunction of the subgoals of interpreted predicates in r. We assume that all subgoals in r have distinct variables in every argument position. Note that this requirement can always be fulfilled by introducing appropriate subgoals in rules using the $=$ predicate. As in the original algorithm, we define a mapping θ that maps each variable in r to a distinct symbol not appearing in $\mathcal{P}$ or r. Instead of evaluating $\mathcal{P}$ with the ground atoms $q_1\theta, \ldots, q_n\theta$, we evaluate $\mathcal{P}$ with facts that are pairs of the form (q, c), where q is ground atom and c is a constraint on the symbols in $Y\theta$. The input to $\mathcal{P}$ will be the pairs $(q_i\theta, c_r\theta)$, for $i = 1, 2, \ldots, n$.

An application of a rule $h :- g_1, \ldots, g_l, c$ proceeds as follows. Let $(a_1, c^1), \ldots, (a_l, c^l)$ be pairs generated previously, such that there is a substitution τ for which $g_i\tau = a_i$ $(1 \leq i \leq l)$. Let c_h be the conjunction $c^1 \wedge \ldots \wedge c^l \wedge c\tau$. If c_h is satisfiable, we derive the pair $(h\tau, c_h)$. In words, the constraint of the new fact generated is the conjunction of the constraints on the facts used in the derivation and the constraints of the rule that was applied in that derivation. We apply the rules of $\mathcal{P}$ until no new pairs are generated. Note that there are only a finite number of possible constraints for the generated facts and, therefore, the bottom-up evaluation must terminate.

Finally, let $(p\theta, c_1), \ldots, (p\theta, c_m)$ be all the pairs generated for $p\theta$ in the evaluation of $\mathcal{P}$; recall that p is the head of Rule (3) and θ is the substitution used to convert the variables of that rule to new symbols. The containment $M(\mathcal{P}) \subseteq M(r)$ holds if and only if $c_r \models c_1 \vee \ldots \vee c_m$, where c_r is the conjunction of interpreted predicates from the body of Rule (3).

Example 4.1: Let $\mathcal{P}_1$ be the program:

$$\begin{array}{llll} r_1: & p(X,Y) & :- & e(X,Z), \ p(Z,Y). \\ r_2: & q(X,Y) & :- & e(X,Y). \end{array}$$

Let $\mathcal{P}_2$ be the program:

$$\begin{array}{llll} s_1: & p(X,Y) & :- & p(X,Z), \ p(Z,Y). \\ s_2: & p(X,Y) & :- & e(X,Y), \ X \leq Y. \\ s_3: & q(X,Y) & :- & e(X,Y), \ Y \leq X. \\ s_4: & q(X,Y) & :- & p(X,Y). \end{array}$$

For a variable X of r, we denote the constant $X\theta$ by x_0. *True* denotes the constraint satisfied by all tuples. To check the uniform containment of r_1, the input to $\mathcal{P}_2$ would be $(e(x_0, z_0), \ True)$ and $(p(z_0, y_0), \ True)$. Rule s_2 will derive $(p(x_0, z_0), \ x_0 \leq z_0)$ and rule s_1 will then derive $(p(x_0, y_0), \ x_0 \leq z_0)$. Since $p(x_0, y_0)$ was only generated under the constraint $x_0 \leq z_0$, we say that rule r_1 is not uniformly contained in $\mathcal{P}_2$.

To check the uniform containment of rule r_2, we begin with $(e(x_0, y_0), \ True)$. Rule s_3 will then

derive $(q(x_0, y_0),\ y_0 \leq x_0)$. Rule s_2 will derive $(p(x_0, y_0),\ x_0 \leq y_0)$ and rule s_4 will use that to derive $(q(x_0, y_0),\ x_0 \leq y_0)$. Since $q(x_0, y_0)$ was derived for both possible orderings of x_0 and y_0, rule r_2 is uniformly contained in $\mathcal{P}_2$. ▮

The correctness of the algorithm is established by the following theorem.

Theorem 4.2: $M(\mathcal{P}) \subseteq M(r) \iff c_r \models c_1 \vee \ldots \vee c_m$.

The theorem is proved by showing the following. Let r be the rule p :– $q_1, \ldots, q_m, c_r$ and Y be the variables appearing in r. If $Y\pi$ is a valid instantiation of the rule r that satisfies c_r, then $p\pi$ is derivable from the database containing the atoms $q_1\pi, \ldots, q_n\pi$ and the program $\mathcal{P}$ if and only if $Y\pi$ satisfies one of $c_1, \ldots, c_m$.

Our bottom-up evaluation of a program with a database containing facts tha are pairs is reminiscent of the procedure used by Kanellakis et al. [KKR90]. In their procedure, an EDB fact may be a generalized tuple specified in the form of a constraint on the arguments of its predicate. However, there is a key difference between the two methods. In [KKR90], every tuple is a constraint *only* on the arguments of the predicate involved. In our procedure, the constraint appearing in a pair is a constraint on all the constants that appear in the database, i.e., all the constants of $Y\theta$, where Y is the set of variables of rule r. Thus, the constraint of a pair may have constants that do not appear in the atom of that pair. The following example illustrates why this difference between the methods is important for detecting uniform containment.

Example 4.3: Consider rules r and s, and let $\mathcal{P}$ consist of rule s.

$$r:\ p(X,Y)\ :-\ q_1(X,Y),\ q_2(U,V).$$
$$s:\ p(X,Y)\ :-\ q_1(X,Y),\ q_2(U,V),\ U \leq V.$$

To show $M(\mathcal{P}) \subseteq M(r)$, we begin with the pairs $(q_1(x_0, y_0),\ True)$ and $(q_2(u_0, v_0),\ True)$, and apply s. If we use the procedure of [KKR90], the result is the pair $(p(x_0, y_0),\ True)$, which has no recording of the fact that its derivation required that $u_0 \leq v_0$. Consequently, we will conclude erroneously that $M(\mathcal{P}) \subseteq M(r)$ holds. In contrast, when our procedure applies rule s to the pairs $(q_1(x_0, y_0),\ True)$ and $(q_2(u_0, v_0),\ True)$, the result is the pair $(p(x_0, y_0),\ u_0 \leq v_0)$, making it clear that s does not contain r, because $True \not\models u_0 \leq v_0$. ▮

The complexity of the algorithm depends on the number of pairs generated during the evaluation of $\mathcal{P}$. In the worst case, it may be exponential in the number of variables of r. A key component in the efficiency of the algorithm is the complexity of checking whether $c_r \models c_1 \vee \ldots \vee c_m$ holds. In [Levy93] we describe how to reduce this problem to a linear programming problem. The result is an algorithm that decides the entailment in time that is polynomial in the size of the disjunction and exponential in the number of $\neq$'s that appear in $c_r, c_1, \ldots, c_m$.

An interesting special case is containment of conjunctive queries with built-in predicates. Klug [Kl88] showed that if all constraints are left-semiinterval or all constraints are right-semiinterval, then containment of conjunctive queries can be decided by finding a homomorphism from one query to the other. For general conjunctive queries, he pointed out that it could be done by finding a homomorphism for every possible ordering of the variables and constants in the queries (recently, van der Meyden [vdM92] has shown that the containment problem of conjunctive queries with order constraints is Π_2^p-complete). In our algorithm, the complexity depends only on the number of orderings that are actually generated during the evaluation of $\mathcal{P}$. More precisely, our algorithm generates partial rather than complete orderings of the variables and constants in the queries. Essentially, it lumps together complete orderings that need not be distinguished from each other in order to test containment. Therefore, our algorithm is likely to be better in practice, albeit not in the worst case (of course, our algorithm also applies to more than just conjunctive queries).

4.1.1 Beyond Uniform Containment

For testing uniform containment of $\mathcal{P}_1$ in $\mathcal{P}_2$, it is enough to check the containment separately for each rule of $\mathcal{P}_1$. Consequently, uniform containment completely ignores possible interactions between the rules, interactions that may imply containment of $\mathcal{P}_1$ in $\mathcal{P}_2$. Consider the following example.

Example 4.4 : Consider the following programs whose query predicate is p. Let $\mathcal{P}_1$ be:

$$r_1:\ p(X)\ :-\ q(X),\ X < 5.$$
$$r_2:\ q(X)\ :-\ e(X),\ X > 0.$$

And let $\mathcal{P}_2$ be the program:

$$r_3:\ p(X)\ :-\ q(X),\ X < 6,\ X > 0.$$
$$r_4:\ q(X)\ :-\ e(X),\ X > 0.$$

The program $\mathcal{P}_1$ is contained in $\mathcal{P}_2$, because whenever $0 < X < 5$, the atom $p(X)$ will be derived from $\mathcal{P}_2$ if $e(X)$ is in the database. However, r_1 is not uniformly contained in $\mathcal{P}_2$ (and, therefore, $\mathcal{P}_1 \not\subseteq^u \mathcal{P}_2$). For example, the model consisting of $\{q(-1), e(-1), \neg p(-1)\}$ is a model of $\mathcal{P}_2$ but not a model of $\mathcal{P}_1$. ▮

The weakness of uniform containment stems from the fact that it considers all models while for proving (ordinary) containment it is sufficient to consider just minimal models.[3] We may, however, try to transform $\mathcal{P}_1$ into an equivalent program $\mathcal{P}'$ with a larger set of models (but, of course, the same set of minimal models, since equivalence must be preserved). One way of doing it is by propagating constraints from one rule to another. The query tree of [LS92] is a tool for doing just that; for the type of constraints considered in this paper the propagation is complete, i.e., each rule ends up having the tightest possible constraint among its variables. In our example, the result of constraint propagation is the following program $\mathcal{P}'$.

$$r_1': \ p(X) \ :- \ q(X), \ X < 5, \ X > 0.$$
$$r_2': \ q(X) \ :- \ e(X), \ X > 0.$$

Now we can show that $\mathcal{P}' \subseteq^u \mathcal{P}_2$, and since $\mathcal{P}_1 \equiv \mathcal{P}'$, it follows that $\mathcal{P}_1 \subseteq^u \mathcal{P}_2$.

4.2 Uniform Equivalence with Stratified Negation

In this section, we describe how to test uniform equivalence of datalog programs with safe, stratified negation. We begin with the case of stratified programs with neither constants nor built-in predicates. By definition, two programs P_1 and P_2 are uniformly equivalent, denoted $P_1 \equiv^u P_2$, if for every database D (that may have both EDB and IDB facts), $P_1(D) = P_2(D)$. Note that applying a stratified program to a database that may also have IDB facts is done stratum by stratum, as in the usual case; in other words, $P(D)$ is the perfect model of the program P and the database D (cf. [Ull88]).

Suppose that that P_1 and P_2 are not uniformly equivalent. Hence, there is a database D_0 such that $P_1(D_0) \neq P_2(D_0)$; D_0 is called a *counterexample*. We may assume that $P_1(D_0) \not\subseteq P_2(D_0)$ (the case $P_2(D_0) \not\subseteq P_1(D_0)$ is handled similarly).

We assume that both P_1 and P_2 have the same set of EDB predicates and the same set of IDB predicates, and moreover, there is a partition of the predicates into strata that is a stratification for both P_1 and P_2. In particular, we assume that the lowest stratum consists of just the EDB predicates and we refer to it as the zeroth stratum. We denote by P_1^i the program consisting of those rules of P_1 with head predicates that belong to the first i strata; similarly for P_2^i. Note that P_1^0 is an empty program (i.e., it has no rules). By definition, $P_1^0(D) = D$ for every database D; similarly for P_2^0.

[3] In our formalism, a set of relations for the EDB and IDB predicates is a minimal model if the IDB part is a minimal model once the EDB facts are added to the program as rules with empty bodies.

We now assume that for some given i, $P_1^i \equiv^u P_2^i$ and we will show how to test whether $P_1^{i+1} \equiv^u P_2^{i+1}$. The algorithm is based on the following two lemmas.

Lemma 4.5: *Suppose that there is an i, such that $P_1^i \equiv^u P_2^i$. If there is a counterexample database D_0, such that $P_1^{i+1}(D_0) \not\subseteq P_2^{i+1}(D_0)$, then there is some rule r of P_1^{i+1} with a head predicate from stratum $i+1$ and a database D, such that*

1. *D is a model of P_2^{i+1} but not a model of r.*
2. *The number of distinct constants in D is no more than the number of distinct variables in r.*

Proof: Let $D' = P_2^i(D_0)$; note that $D' = P_2^i(D')$. By the assumption in the lemma, $P_1^i(D_0) = P_2^i(D_0)$ and, hence, D' is also a counterexample, i.e., $P_1^{i+1}(D') \not\subseteq P_2^{i+1}(D')$. Now let $\bar{D} = P_2^{i+1}(D')$. Observe that $\bar{D}$ and D' have the same set of facts for predicates of the first i strata, since $D' = P_2^i(D')$. In addition, observe that $D' \subseteq \bar{D}$. These observations imply that $P_1^{i+1}(D') \subseteq P_1^{i+1}(\bar{D})$. Thus, $P_1^{i+1}(\bar{D}) \not\subseteq P_2^{i+1}(\bar{D})$, because $P_1^{i+1}(D') \not\subseteq P_2^{i+1}(D')$ and $P_2^{i+1}(D') = P_2^{i+1}(\bar{D})$.

So, we have shown that $P_1^{i+1}(\bar{D}) \not\subseteq P_2^{i+1}(\bar{D})$ and $\bar{D}$ is a model of P_2^{i+1}. Therefore, there is a rule r in P_1^{i+1} of the form

$$h :- q_1, \ldots, q_m, \neg s_1, \ldots, \neg s_l$$

and a substitution θ, such that

- the predicate of h is from stratum $i+1$,
- θ is a mapping from the variables of r to constants,
- $q_i\theta \in \bar{D}$ $(1 \leq i \leq m)$,
- $s_j\theta \notin \bar{D}$ $(1 \leq j \leq l)$, and
- $h\theta \notin \bar{D}$.

The above and the fact $\bar{D} = P_2^{i+1}(\bar{D})$ imply that the database $\bar{D}$ is a model of P_2^{i+1} but not of r^{i+1}.

Let D be the database consisting of facts from $\bar{D}$ that have only constants from $r\theta$. Database D is also a model of P_2^{i+1}. In proof, suppose that D is not a model of P_2^{i+1}. Thus, there is a rule $\bar{r}$ of P_2^{i+1} and a substitution τ, such that

1. the head $\bar{h}$ of $\bar{r}$ satisfies $\bar{h}\tau \notin D$,
2. every positive subgoal $\bar{q}$ of $\bar{r}$ satisfies $\bar{q}\tau \in D$, and
3. every negative subgoal $\bar{s}$ of $\bar{r}$ satisfies $\bar{s}\tau \notin D$.

By the definition of D, if g is a ground fact having only constants from D, then $g \in D$ if and only if $g \in \bar{D}$; moreover, for every negative subgoal $\bar{s}$, the constants appearing in $\bar{s}\tau$ are all from D, since rules

are safe (cf. [Ull88]). Therefore, items (1)–(3) hold even if we replace D with $\bar{D}$, and so it follows that $\bar{D}$ is not a model of $\bar{r}$—a contradiction, since $\bar{D}$ is a model of P_2^{i+1} and $\bar{r}$ is a rule of P_2^{i+1}. Thus, we have shown that D is a model of P_2^{i+1}. Furthermore, items (1)–(3) above imply that D is not a model of r. So, the lemma is proved. ▮

Lemma 4.6: *Suppose that $P_1^i \equiv^u P_2^i$. Moreover, suppose that there is a database D that is a model of P_2^{i+1} and is not a model of some rule r of P_1^{i+1} having a head predicate from stratum $i+1$. Then $P_1(D) \not\subseteq P_2(D)$ and, hence, $P_1 \not\equiv^u P_2$.*

Proof: From the assumptions in the lemma, it follows that rule r can be applied to D to generate a new fact g that is not already in D. Note that $g \notin P_2(D)$, since $P_2^{i+1}(D) = D$ and strata higher than $i+1$ do not include facts with the same predicate as that of g. If we show that rule r can still generate g even when P_1 is applied to D, it would follow that $g \in P_1(D)$, and hence, $P_1(D) \not\subseteq P_2(D)$. To show that, recall that $P_1^i \equiv^u P_2^i$ and D is a model of P_2^{i+1}; therefore, D is also a model of P_1^i. Thus, rule r can still generate g during the application of P_1 to D, since nothing is generated by rules of lower strata. ▮

The algorithm of Figure 2 tests whether $P_1 \equiv^u P_2$; its correctness follows from the above two lemmas and the following proposition.

Proposition 4.7: *$P_1(D) \not\equiv P_2(D)$ if and only if there is some i and a database D, such that either $P_1^i(D) \not\subseteq P_2^i(D)$ or $P_2^i(D) \not\subseteq P_1^i(D)$.*

Note that in the algorithm, it does not matter what are the constants in S as long as their number is equal to the number of distinct variables in the given rule r. Also, if two databases over constants from S are isomorphic, it is sufficient to consider just one of them.

Example 4.8: Let P_1 consist of the rules:

$$
\begin{aligned}
r_1 &: Iown(X,Y) \;:-\; own(X,Y). \\
r_2 &: Iown(X,Y) \;:-\; lives(X,Z),\ inHouse(Z,Y). \\
r_3 &: Iown(X,U) \;:-\; own(X,Z),\ lives(Y,Z),\ Iown(Y,U). \\
r_4 &: buys(X,Y) \;:-\; likes(X,Y),\ \neg Iown(X,Y).
\end{aligned}
$$

Let P_2 consist of the rules r_1, r_4 and the rule:

$$r_5 : Iown(X,Y) \;:-\; Own(X,Z),\ inHouse(Z,Y).$$

The EDB relation *own* describes an ownership relationship between persons and objects. The IDB

```
procedure check(P1, P2);
begin
    for every rule r of P1 do
    begin
        Let S be a set of v distinct constants,
        where v is the number of variables in r;
        for every database D that includes
            only constants from S do
            if D is a model of P2 but not of r
            then return false;
    end;
    return true;
end;
begin /* main procedure */
    for i := 1 to max-stratum do
        if not check(P1^i, P2^i) or not check(P2^i, P1^i)
        then return P1 ≢^u P2;
    return P1 ≡^u P2;
end.
```

Figure 2: An algorithm for testing $P_1 \equiv^u P_2$.

relation *Iown* represents a landlord's perspective of the ownership relation. The programs P_1 and P_2 are not uniformly equivalent. Specifically, consider the database D_0:

$$\{likes(a,o),\ lives(b,h),\ own(b,o),\ own(a,h)\}$$

Rule r_4 (of P_2) and program P_1 satisfy $r_4(D_0) \not\subseteq P_1(D_0)$, since $Iown(a,o) \notin r_4(D_0)$ and therefore $buys(a,o) \in r_4(D_0)$, while the converse is true for $P_1(D_0)$. ▮

To extend the algorithm to programs with built-in predicates (and constants), we need to check for the possibility that a database may become a counterexample by analyzing the built-in constraints. One conceptually simple (albeit not the most efficient) way of doing it is by using the algorithm of Figure 2, but with the following modifications. Let C be the set of constants appearing in either P_1 or P_2. Instead of considering every database over constants from S, we should consider every database over constants from $S \cup C$. Moreover, for each database we should consider every total order on the constants of the database, such that the order is consistent with any order that may implicitly be defined on C (e.g., if C is a set of integers, then presumably the usual order on integers should apply to C). For each such database and total order defined on its constants, we should apply the given test of the *check* procedure; that is, we should test whether D is a model of P_2 but not of r. The rest of the algorithm is the same as earlier. Thus, we get the following result; for the full details of the

proof and for a more efficient algorithm than the one described above see [Levy93].

Theorem 4.9 : *Uniform equivalence for datalog programs with safe, stratified negation and built-in predicates is decidable.*

5 Concluding Remarks

We have presented an analysis of the notion of independence and described algorithms for detecting independence of queries from updates. Our formulation of the problem gives us flexibility in the analysis. For example, we can distinguish between the case in which an updates is specified intensionally and the actual tuples to be inserted are computed at update time, and the case in which the set of tuples to be inserted is given a priori. Our framework and algorithms can also be extended to incorporate integrity constraints, as in Elkan [El90].

Posing the problem of independence as a problem of equivalence suggests that further algorithms for independence can be found by trying to identify additional sufficient conditions for equivalence. One possibility mentioned in this paper involves program transformations that increase the set of models but preserve the set of minimal models. Consequently, these transformations increase the possibility of detecting equivalence by an algorithm for uniform equivalence. More powerful transformations can be obtained by considering, for example, only minimal derivations [LS92].

In this paper, we have considered the problem of detecting independence assuming we have no knowledge of the EDB relations. An important problem, investigated in [BCL89] and [GW93] is detecting independence when some of the EDB relations are known and can be inspected efficiently. Combining our techniques with the ones described in those papers is an intersting area for future research.

6 Acknowledgments

The authors thank Ashish Gupta for many helpful discussions and Jose Blakeley for very useful comments regarding the presentation of the material.

References

[BCL89] Blakeley, J. A., N. Coburn, Larson, P. A.: Updating derived relations: detecting irrelevant and autonomously computable updates. *Transactions of Database Systems, Vol 14, No. 3, pp. 369-400, 1989*

[El90] Elkan, C.: Independence of Logic Database Queries and Updates. *Proc. 9th ACM Symp. on Principles of Database Systems,* 1990, pp. 154–160.

[GW93] Gupta A., Widom J.: Local Verification of Global Integrity Constraints in Distributed Databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* 1993, pp. 49–58.

[KKR90] Kanellakis, P.C., Kuper, G.M., and Revesz, P.Z.: Constraint query languages. *Proc. 9th ACM Symp. on Principles of Database Systems,* 1990, pp. 299–313.

[Kl88] Klug, A.: On conjunctive queries containing inequalities. *JACM,* Vol. 35, No. 1, 1988, pp. 146–160.

[Levy93] Levy, A.: Irrelevance reasoning in knowledge based systems. *Forthcoming Ph.D thesis, Stanford University, 1993.*

[LS92] Levy, A. and Sagiv, Y.: Constraints and redundancy in datalog. *Proc. 11th ACM Symp. on Principles of Database Systems,* 1992, pp. 67-80.

[LMSS93] Levy, A.Y., Mumick, I.S, Sagiv, Y. and Shmueli, O.: Query-Reachability and Satisfiability in Datalog. To appear in *Proc. 12th ACM Symp. on Principles of Database Systems,*, 1993.

[Sa88] Sagiv, Y.: Optimizing datalog programs. In *Foundations of Deductive Databases and Logic Programming,* (J. Minker, ed.), Morgan Kaufmann Publishers, 1988, pp. 659–698.

[Sh87] Shmueli, O.: Decidability and expressiveness aspects of logic queries. *Proc. 6th ACM Symp. on Principles of Database Systems,* 1987, pp. 237–249.

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

[vdM92] van der Meyden R.: The Complexity of Querying Indefinite Data about Linearly Ordered Domains. *Proc. 11th ACM Symp. on Principles of Database Systems,* 1992, pp. 331–345.