March 1986 Report No. STAN-CS-86-1132

Optimizing Datalog Programs

by

Yehoshua Sagiv

Department of Computer Science

Stanford University
Stanford, CA 94305

OPTIMIZING DATALOG PROGRAMS

Yehoshua Sagiv†
Stanford University

ABSTRACT

Datalog programs, i.e, Prolog programs without function symbols, are considered. It is assumed that a. variable appearing in the head of a. rule must also appear in the body of the rule. The input of a program is a set of ground atoms (which are given in addition to the program's rules) and, therefore, can be viewed as an assignment of relations to some of the program's predicates. Two programs are equivalent if they produce the same result for all possible assignments of relations to the estensional predicates (i.e., the predicates that do not appear as heads of rules). Two programs are uniformly equivalent if they produce the same result for all possible assignments of initial relations to all the predicates (i.e., both extensional and intentional). The equivalence problem for Datalog programs is known to be undecidable. It is shown that uniform equivalence is decidable, and an algorithm is given for minimizing a Datalog program under uniform equivalence. A technique for removing parts of a program that axe redundant under equivalence (but not under uniform equivalence) is developed. A procedure for testing uniform equivalence is also developed for the case in which the database satisfies some constraints.

I. Introduction

Horn-clause programs without function symbols, also known as *Datalog* programs, are an important part of *deductive* (or *logical)* databases (Gallaire and Minker [1978]). Recent works has-e addressed the problem of finding efficient evaluation methods for queries expressed as Datalog programs‡ (e.g., Bancilhon et al. [1986a], Henschen and Naqvi [1984], Iiifer and Lozinskii [1986], Lozinskii [1985], McKay and Shapiro [1981], Rohmer and Lescoeur [1985], Saccà and Zaniolo [1986], Ullman [1985], Van Gelcler [1986]). It is important to remember that database applications usually require finding all the answers to a query, and when there are no function symbols, it is possible to find all the answers in finite time by a naive bottom-up computation. However, that requires retrieving a.11 the tuples of the relations specified in the program. Therefore, a common theme in many of the proposed methods for efficient query evaluation is to use the constants specified in the query in order to restrict the size of intermediate results as soon as possible.

In this paper we take a complementary approach to the one just mentioned, and investigate the question of how to remove redundant parts from a Datalog program. A redundant part in a program is either a redundant rule or a redundant atom in the body of a rule. In most cases, removing redundant parts can only reduce the time needed to evaluate the query, because it reduces the number of joins done during the evaluation. For example, if the query is going to be computed the "magic set" method of Bancilhon

† On a leave of absence from Hebrew University. Work supported by a grant of AT&T Foundation, a grant of IBM Corp., and an NSF grant IST-W-12791.

‡ See also Bancilhon and Ramakrishnan [1986b] for a thorough review of this subject.

et al. [1986a], then removing redundant parts can only speed up the computation. Some works (e.g., Chakravarthy et al. [1986], Finger [1986], King [1981]) have also considered the opposite approach, namely, optimizing a program by adding more conjuncts to the body of a rule. This is usually useful when it is required to find only one answer to a query (as opposed to finding all the answers). In some cases, it could be useful even if all the answers are required. For example, if the intersection of two relations is to be computed and the database has a third relation that contains this intersection, then it may be better to compute the intersection of all three relations rather than just the original two (whether it is indeed better depends upon the sizes of the three relations, the size of their intersection, and the available indices). As already said, in this paper we develop methods for finding redundant parts of a program and removing them. Our ideas, however, can also be used to determine when a redundant atom can be added to the body of a rule and, therefore, they can also be incorporated in the type of optimization that adds conjuncts rather than removes them.

Usually, a Datalog program gets as input relations for the extensional predicates, namely, those predicates that do not appear as heads of rules, and the answer consists of relations for the intentional predicates, namely those that appear as heads of rules. The process of optimization requires finding a program of least cost, which is equivalent to the original one, that is, for all possible inputs, the optimized program and the original one have the same output. However, the equivalence problem for Datalog programs is undecidable (Shmueli [1986]), as is the problem of whether a program has a redundant rule Gaifman [1986]).

We propose the notion of *uniform* equivalence which is defined as follows. Two programs are uniformly equivalent if they have the same output for all inputs, where a possible input for a program is an assignment of initial relations to the extensional as well as the intentional predicates, and the program computes the final relations for the intentional predicates.† Clearly, uniform equivalence implies equivalence, but the converse is not true. We show how to minimize a program under uniform equivalence; that is, we give an algorithm that removes all redundant rules and all redundant atoms from the remaining rules while preserving uniform equivalence. The algorithm has an exponential running time in the worst case, but the time is esponential only in the size of the program, which is typically much smaller than the size of the database. Therefore, minimizing a program is expected to reduce the total time spent on optimization and evaluation. As an example, given- the rule

$$G(x,y,z) \text{ :- } G(x, w, z),\ A(w, y), A(w, z), A(z, z), A(z, y).$$

our algorithm determines that the atom $A(w, y)$ is redundant and, consequently, the rule can be replaced with

$$G(x, y, z) \text{ :- } G(x, w, z), A(w, z), A(z, z), A(z, y).$$

To fully appreciate the importance of the algorithm, one should realize that optimization under uniform equivalence, is the only one that can be done locally. In comparison, if

† Clearly, for each intentional predicate, the final relation contains the initial one, and for each extensional predicate, the final relation is the same as the initial one.

a subset of the rules of a program P is replaced with an equivalent (but not uniformly equivalent) subset of rules, then the resulting program is not necessarily equivalent to P.

We also give a technique for minimizing Datalog programs under equivalence. Since this is an undecidable problem, it is clear that our technique may not find all the parts of a program that are redundant under equivalence. However, we believe that this technique is going to be useful in many practical situations. For example, our technique can easily show that in the program

$G(x,z)$:- $A(x,z)$.
$G(x,z)$:- $G(x,y)$, $G(y,z)$, $A(y,w)$.

the atom $A(y, w)$, in the second rule, is redundant and can be removed without changing the result.

II. Basic Definitions

We consider *Datalog* programs, i.e., Prolog programs having only predicates, variables and constants. Function symbols as well as other features of Prolog (e.g., lists, cuts, arithmetic operations) are not permitted. A Datalog *program* is a set of rules (also known as *Horn-clause* rules). Each rule has a head, appearing on the left-hand side of the symbol :-, and a body, appearing on the right-hand side of the symbol :-. The head of a rule is a single atomic formula or simply *atom,* that is, a *predicate*† with either a variable or a constant in each argument position. We assume that constants are integers. For example, $Q(x, y, 3.10)$ is an atom, where Q is a predicate, x and y are variables and 3 and 10 are constants. The body of a rule is a conjunction of atoms.

Example 1: The following is a program with two rules.

$G(x, z)$:- $A(x, z)$.
$G(x, z)$:- $G(x, y)$, $G(y, r)$.

The input to this program is A and the output, G, is the transitive closure of A. □

We conveniently denote a conjunction of atoms (e.g., in the body of the second rule) by separating the atoms with commas rather than with the "logical and" symbol $\wedge$.

We do not consider rules with an empty head; rules of these form are used in logical databases to express integrity constraints.

We assume that every variable in the head of a rule must also appear in the body (therefore, rules with an empty body are not allowed unless the head has only constants and no variables). Thus, if we want to write rules for the predicate $Anc(x, y)$, whose meaning is that x is an ancestor of y, we cannot write the rule $Anc(x, x)$:-, whose meaning is that everybody is his own ancestor. This does not present any real restriction, as far as databases are concerned, since each variable is assumed to be bound to a finite set. For example, in the rule $Anc(x, x)$:-, variable x is bound to the set of all persons mentioned in the database. Thus, the rule $Anc(x,x)$:- can be replaced with $Anc(x,x)$:- *Person(x).*

† In traditional database terminology, a predicate is called a relation scheme.

III. Computing a Program

A relation q for a predicate Q is a set of *ground* atoms of Q, i.e., atoms having only constants (and no variables), e.g., $Q(34,3,12,15)$. Unless otherwise stated, we assume that relations are finite. A collection of relations, such as a database, can be viewed as a single set consisting of all the ground atoms of these relations. If $q_1, \ldots, q_n$ are relations for the predicates $Q_1, \ldots, Q_n$, respectively, then $\langle q_1, \ldots, q_n \rangle$ denotes their union,† namely, the set containing the ground atoms of all the q_i. The set $\langle q_1, \ldots, q_n \rangle$ is also called an *interpretation* or a *structure.*

A predicate is *intentional*, in a given program *P*, if it appears as the head of some rule in *P*. An intentional predicate is supposed to be evaluated by the program *P*. A predicate is *extensional* if it does not appear in the head of any rule.

A program *P* has an associated directed graph, called the dependence graph, that has a node for each predicate of the program, and an edge from predicate Q to predicate *R* whenever predicate Q is in the body of some rule and predicate *R* is in the head of that same rule. Program *P* is recursive if its dependence graph has a cycle. A predicate Q is recursive in program *P* if there is a path from Q to itself. Note that recursive predicates are intentional, but an intentional predicate is not necessarily recursive. Finally, a rule is *recursive* if the dependence graph has a cycle that includes the predicate from the rule's head and a predicate from the rule's body. In particular, a rule is recursive if the predicate in the rule's head appears also in the body.

The *input* for a program *P* is a relation for each extensional predicate, and it is called the *extensional* database (abbr. EDB). The output computed by *P* is, in principle, a relation for each intentional predicate, and it is called the *intentional database* (abbr. IDB). To simplify notation, we formally define the output to be both the EDB and the IDB, and simply call it the database (abbr. DB). Note that the EDB-part of the output is the same as the input.

Now we are going to describe how the output can be computed. The ground atoms of the DB are known facts. Initially, the known facts are those in the EDB. A program's rule states that if some facts are known, then another fact can be deduced from them. Newly deduced facts become ground atoms of the IDB (and hence of the DB) and, so, the rules can be used once again to deduce more new facts. Formally, a rule r is used to deduce a new fact *by instantiating* its variables to constants, i.e., substituting a constant for all occurrences of each variable. If under the instantiation, each atom in the body of rule r becomes a ground atom of the DB, then the instantiated head of the rule is added to the IDB.

Example 2: Consider the program of Example 1 and suppose that the EDB is the following set of ground atoms: $\{A(1, 2), A(1,4), A(4,1)\}$. Initially, the IDB is the empty set. If, in the first rule, we instantiate x to 1 and z to 2, then the body of the rule becomes $A(1,2)$, which is a ground atom of the EDB. Therefore, $G(1,2)$ is added to the IDB. Two similar. instantiations of the first rule add $G(1,4)$ and $G(4,1)$ to the IDB. As for the second rule, the instantiation of x to 1, y to 4, and z to 1 produces the ground atoms $G(1,4)$ and

† Note that this definition of the union is more general than the definition of the union operator in relational algebra.

$G(4, 1)$ in the hotly of the rule. Since both are already in the DB, the instantiated head, $G(1, 1)$, is added to the IDB. Similarly, instantiating both x and z to 4 and y to 1 yields $G(4,4)$. Finally, $G(4,2)$ is obtained when x is instantiated to 4, y to 1, and z to 2. No more ground atoms can be produced by any instantiation, and so, the DB

$$\{A(1,2), A(1,4), A(4, 1), G(1, 2), G(1,4), G(4,1), G(1, 1), G(4,4), G(4,2)\}$$

is the output of the program for the above EDB. •I

If the input for a program consists of finite relations, then the output is also a. set of finite relations. Computing the output by repeatedly instantiating rules, until no new ground atoms can be generated, is known as *bottom-up* computation. For a fixed program, this method runs in polynomial time in the size of the EDB.

Let P be a program with the extensional predicates $E_1, \ldots, E_n$ and the intentional predicates $I_1, \ldots, I_m$. Given an EDB $\langle e_1, \ldots, e_n\rangle$, where each e_k is a relation for E_k, the DB computed by P is denoted by $P(\langle e_1, \ldots, e_n\rangle)$. Recall that $P(\langle e_1, \ldots, e_n\rangle)$ is a set of ground atoms, and the EDB-part of $P(\langle e_1, \ldots, e_n\rangle)$ is the same as the input.

Sometimes we would like to view P as a program whose input is both an EDB and an IDB. The output is computed as defined earlier, i.e., by repeatedly instantiating rules until no new ground atoms can be added to the IDB. Clearly, the output is a DB that contains the input. When P is viewed as a program whose input is both an EDB $\langle e_1, \ldots, e_n\rangle$ and an IDB $\langle i_1, \ldots, i_m\rangle$, the output of P is denoted by $P(\langle e_1, \ldots, e_n, i_1, \ldots, i_m\rangle)$.

Example 3: Let P be the program of Example 1. In Example 2, we have computed the output of P for the input. $\{A(1,2), A(1, 4), A(4, 1)\}$. It is easy to see that the output of P for the input, $\{A(1,2), A(1, 4), G(4,1)\}$ is the same as the one computed in Example 2, but with the ground atom $A(4, 1)$ omitted. •1

IV. Equivalence, Uniform Equivalence, and Models

Let P_1 and P_2 be programs with the same set of extensional predicates and the same set of intentional predicates. Program P_1 contains P_2, written $P_2 \subseteq P_1$, if for a.11 EDBs $\langle e_1, \ldots, e_n\rangle$, the output of P_1 contains that of P_2, i.e., $P_2(\langle e_1, \ldots, e_n\rangle) \subseteq P_1(\langle e_1, \ldots, e_n\rangle)$. In traditional database terminology, it means that for each predicate Q, the relation for Q in the DB $P_2(\langle e_1, \ldots, e_n\rangle)$ is a subset of the relation for Q in the DB $P_1(\langle e_1, \ldots, e_n\rangle)$.

Program P_1 and P_2 are *equivalent*, written $P_2 \equiv P_1$, if $P_2 \subseteq P_1$ and $P_1 \subseteq P_2$. Equivalence simply means that the two programs have the same output whenever they are given the same EDB as input.

Program P_1 *uniformly contains* P_2, written $P_2 \subseteq^u P_1$, if for all pairs of an EDB $\langle e_1, \ldots, e_n\rangle$ and an IDB $\langle i_1, \ldots, i_m\rangle$, the following containment holds:

$$P_2(\langle e_1, \ldots, e_n, i_1, \ldots, i_m\rangle) \subseteq P_1(\langle e_1, \ldots, e_n, i_1, \ldots, i_m\rangle)$$

Program P_1 and P_2 are *uniformly equivalent*, written $P_2 \equiv^u P_1$, if $P_2 \subseteq^u P_1$ and $P_1 \subseteq^u P_2$. Uniform equivalence means that the two programs have the same output whenever they are given the same input, where the input may also include ground atoms for some intentiona. predicates.

Proposition 1: Uniform containment implies containment.

Proof: If $P_2(\langle e_1, \ldots, e_n, i_1, \ldots, i_m\rangle) \subseteq P_1(\langle e_1, \ldots, e_n, i_1, \ldots, i_m\rangle)$ for all $\langle e_1, \ldots, e_n\rangle$ and $\langle i_1, \ldots, i_m\rangle$, then in particular, for all EDBs $\langle e_1, \ldots, e_n\rangle$

$$P_2(\langle e_1, \ldots, e_n, \emptyset, \ldots, \emptyset\rangle) \subseteq P_1(\langle e_1, \ldots, e_n, \emptyset, \ldots, \emptyset\rangle)$$

where $\emptyset$ denotes the empty relation. Therefore, $P_2(\langle e_1, \ldots, e_n\rangle) \subseteq P_1(\langle e_1, \ldots, e_n\rangle)$ for all EDBs $\langle e_1, \ldots, e_n\rangle$ and, so, $P_2 \subseteq P_1$. $\square$

Example 4: This example shows that equivalence does not always imply uniform equivalence. Let P_1 be the program of Example 1, and let P_2 be the following program.

$G(x, z)$:- $A(x, z)$.
$G(x, z)$:- $A(x, y)$, $G(y, z)$.

Both programs compute the transitive closure of A when the input has only ground atoms of A, i.e., they are equivalent. Moreover, as we shall see later, P_1 uniformly contains P_2. But P_2 does not uniformly contain P_1. To see this, suppose that the input is the empty relation for A and some nonempty relation g for G, such that g is not the transitive closure of itself. Then, the output of P_2 is the same as the input, i.e., g, while the output of P_1 is the transitive closure of g. Thus, containment does not always imply uniform containment. $\square$

A DB $\langle e_1, \ldots, e_n, i_1, \ldots, i_m\rangle$ is a model of P if

$$\langle e_1, \ldots, e_n, i_1, \ldots, i_m\rangle = P(\langle e_1, \ldots, e_n, i_1, \ldots, i_m\rangle)$$

that is, no new ground atoms are generated when the program P is applied to the given DB. Let $M(P)$ denote the set of all models of P. It is well known that the set $M(P)$ is closed under intersection, and the output of P, given an input $\langle e_1, \ldots, e_n, i_1, \ldots, i_m\rangle$, is the minimal model of P that contains the input (Van Emden and Kowalski [1976]).

The above results imply that two programs are equivalent if they have the same set of models that are minimal with respect to the IDB.† Uniform equivalence is similarly characterized in terms of models. Programs P_1 and P_2 are uniformly equivalent if they have the same set of models, i.e., $M(P_1) = M(P_2)$. The following proposition, whose proof is given in the appendix, is a characterization of uniform containment. Note that uniform containment means containment of the sets of models in the opposite direction.

Proposition 2: $P_2 \subseteq^u P_1 \iff M(P_1) \subseteq M(P_2)$.

When discussing uniform containment of two programs P_1 and P_2, it is not necessary to assume that they have the same set of predicates, provided that an input is any set of ground atoms for the predicates appearing in either P_1 or P_2. It is even possible for an extensional predicate in one program to be an intentional predicate in the other program.

† A model d is minimal with respect to the IDB if there is no proper subset d' of d which is also a model and has the same EDB as d.

Example 5: Let P_1 be the program of Example 1, and let P_2 be obtained form P_1 by adding the rule $A(x, z)$:- $A(x, y), G(y, z)$. Note that program P_2 has only intentional predicates and, so, it is meaningful only if it may have a nonempty IDB as input. Since every rule of P_1 is also a rule of P_2, it is easy to verify that $P_1(d) \subseteq P_2(d)$ for every DB d consisting of ground atoms for A and G. Therefore, there is uniform containment, i.e., $P_1 \subseteq^u P_2$. $\Box$

We conclude this section with a. simple observation that further illustrates the relationship between containment and uniform containment, and shows that sometimes equivalence implies uniform equivalence. Given programs P_1 and P_2, we can construct programs P_1' and P_2' such that, $P_2 \subseteq^u P_1$ if and only if $P_2' \subseteq P_1'$. The programs P_1' and P_2' are obtained by adding rules that give arbitrary initial values to the intentiona. predicates. The rule added for an intentional predicate† $B(x_1, \ldots, x_n)$ is simply $B(x_1, \ldots, x_n)$:- $B_0(x_1, \ldots, x_n)$, where B_0 is a, predicate that, does not appear in any other rule. Note that if P_1 and P_2 already have a rule of the form $B(x_1, \ldots, x_n)$:- $G(x_1, \ldots, x_n)$, where G appears only in this rule, then there is no need to add the rule for B. In particular, if for every intentional predicate B, both P_1 and P_2 already have a. rule of the form $B(x_1, \ldots, x_n)$:- $G(x_1, \ldots, x_n)$, where G appears only in this rule, then $P_2 \subseteq^u P_1$ if and only if $P_2 \subseteq P_1$.

V. Optimizing Recursive Programs

In this paper we consider a particular type of optimization, namely, removing redundant atoms from the body of a rule, and removing redundant rules from a program. This optimization is useful, since it reduces the number of joins needed to compute the output. The problem of optimizing non-recursive programs has been solved, both for the case of single-rule programs (Aho, Sagiv and Ullman [1979], and Chandra and Merlin [1976]) and for the case of programs wit.11 many rules (Sagiv and Yannakakis [1980]). Essentially, it has been shown that a program consisting of non-recursive rules has a unique equivalent program with a minima.1 number of rules and a minimal number of atoms in the body of each rule. Optimizing recursive programs, however, is considerably harder. In fact, there can be no algorithm for optimizing recursive programs, since equivalence of recursive programs is undecidable (Shmueli [1986]), as is the problem of whether a rule is redundant in a given recursive program, i.e., the problem of whether a rule can be deleted while preserving equivalence (Gaifman [1986]). These undecidability results are valid even if we consider only linear programs, i.e., programs in which the body of each rule has at most one recursive predicate.

In Section VII, we shall show that recursive programs can be optimized under uniform equivalence, i.e., we shall give an algorithm that finds redundant atoms in the body of a rule, as well as redundant rules in a. program, and removes them while preserving uniform equivalence. The final result is a program with neither an atom nor a rule that can be deleted while preserving uniform equivalence. Unlike the non-recursive case, however, the final result of this optimization is not necessarily unique (i.e., it may depend upon the order in which atoms and rules are considered for deletion).

In Section XI, we shall describe a more elaborate procedure that can remove redundant

† If B is intentional only in one of the programs P_1 and P_2, then we still add the rule for B to both of them.

atoms while preserving equivalence, but not uniform equivalence. This procedure is not, completely algorithmic, since it has to use some simple heuristics: and, of course, it cannot always remove all atoms that are redundant under equivalence. We believe, however, that this procedure is going to be an important tool for optimizing Datalog programs.

VI. Testing Uniform Containment and Equivalence

Testing $M(P_1) \subseteq M(P_2)$ (and, by Proposition 2, testing $P_2 \subseteq^u P_1$) is conceptually simple and can be clone by the *chase* process (Maier et al. [1979]). Originally the chase n-as used to test implications of database dependencies, and recently it has been noted by Cosmadakis and Kanellakis [1986] that the chase can also test uniform containment of Datalog programs with only one predicate. It is easy to see, however, that the same is true for programs with many predicates.

Note that $M(P_1) \subseteq M(P_2)$ ifand only if for all rules r of P_2, $M(P_1) \subseteq M(r)$, because a DB d is a model of P_2 if and only if it is a model of every rule r of P_2 . Therefore, by Proposi tion 2, $P_2 \subseteq^u P_1$ if and only if for all rules r of P_2, $r \subseteq^u P_1$ (note that r is a single rule program).

We will now show how to test $r \subseteq^u P_1$, where r is a single rule. Let r be the rule h :- b, i.e., h is the head and b is the bocly. In order to test whether r $\subseteq^u P_1$, we have to consider the atoms of b as an input DB for P_1. Technically, the atoms of b are converted into a DB by substituting distinct constants for the variables of r and, as a result, b becomes a. set of ground atoms. The uniform containment $r \subseteq^u P_1$ holds if and only if P_1 $(b\theta)$ contains the ground atom $h\,\theta$, where

1. θ is a one-to-one substitution that maps each variable of r to a distinct constant that is not already in r,
2. $b\theta$ is the set of ground atoms obtained from b by substituting according to θ, and
3. $h\theta$ is the ground atom obtained from h by substituting according to θ.

Example 6: Consider again programs P_1 and P_2 of Examples 1 and 4. Recall that P_1 is

$G(x, z)$:- $A(x, z)$.
$G(x, z)$:- $G(x, y), G(y, z)$.

and P_2 is

$G(x, z)$: - $A(x, z)$.
$G(x, z)$:- $A(x, y), G(y, z)$.

We are now going to show that $P_2 \subseteq^u P_1$. First, the variables of P_2 have to be instantiated to distinct constants. In order that the instantiation be visible throughout the example, we use the subscript **0** to denote constants, that is, variable x is instantiated to a constant denoted by x_0, variable y is instantiated to the constant y_0, and z to z_0. The rules of P_2 must be considered one by one. So, consider the first rule of P_2, denoted r_1, i.e.,

$G(x, z)$:- $A(x, z)$.

The instantiated body of r_1 is the DB $\{A(x_0, z_0)\}$. When P_1 is applied to this DB, the result is $\{G(x_0, z_0), A(x_0, z_0)\}$. Since the result contains the instantiated head of r_1 , it follows that $r_1 \subseteq^u P_1$.

Now consider the second rule of P_2 , denoted r_2, i.e.,

$$G(x, z) \text{ :- } A(x, y), G(y, z).$$

The instantiated body of r_2 is the DB $\{ A(x_0, y_0), G(y_0, z_0)\}$. The first rule of P_1 can be applied to $A(x_0, y_0)$ to produce $G(x_0, y_0)$, and then an application of the second rule gives $G(x_0, z_0)$. Thus, the instantiated head of r_2 (i.e., $G(x_0, z_0)$) is in the result, and so. $r_2 \subseteq^u P_1$. Since P_1 uniformly contains every rule of P_2, it follows that $P_2 \subseteq^u P_1$.

Nest,, we will show that $P_1 \not\subseteq^u P_2$. Consider the second rule of P_1, denoted s, i.e.,

$$G(x, z) \text{ :- } G(x, y), G(y, z).$$

The instantiated body is the DB $\{ G(x_0, y_0), G(y_0, z_0)\}$. No new ground atoms are produced when P_2 is applied to this DB. Therefore, $s \not\subseteq^u P_2$ and, so, $P_1 \not\subseteq^u P_2$. •I

Example 7: Let, P_1 *be* the program

$$G(x, y, z) \text{ :- } G(x, w, z),\ A(w, y),\ A(w, z),\ A(z, z),\ A(z, y).$$

and P_2 be the program

$$G(x, y, z) \text{ :- } G(x, w, z), A(w, z), A(z, z), A(z, y).$$

Since the body of the rule of P_2 is a subset of the body of the rule of P_1, it is clear that $P_1 \subseteq^u P_2$. We will show that $P_2 \subseteq^u P_1$ and, hence, $P_1 \equiv^u P_2$. The instantiated body of the single rule of P_2 is the DB

$$\{G(x_0, w_0, z_0),\ A(w_0, z_0),\ A(z_0, z_0),\ A(z_0, y_0)\}$$

We first apply P_1 to this DB by instantiating the variables of P_1 as follows. Variable x is instantiated to x_0, variable w to w_0, and both z and y to z_0. It is easy to check that, under this instantiation, the body of the rule of P_1 becomes a. subset of the DB and, therefore, the ground a tom $G(x_0, z_0, z_0)$ is added to the DB. Now we apply P_1 again by instantiating x to x_0, both w and z to z_0, and y to y_0, and the result is $G(x_0, y_0, z_0)$, which is the instantiated head of the rule of P_2. Therefore, $P_2 \subseteq^u P_1$. □

VII. Minimizing Programs Under Uniform Equivalence

Having an algorithm for testing uniform equivalence makes it possible to optimize Datalog programs in two ways. The first one has just been illustrated in Example 7, and it involves eliminating redundant atoms from the body of a rule in the following way. Consider a rule r and let $\hat{r}$ be the result of deleting one of the atoms in the body of r. If $\hat{r} \subseteq^u r$, then rule r can be replaced with $\hat{r}$, since it follows that $\hat{r} \equiv^u r$ (note that $r \subseteq^u \hat{r}$ is trivially true). When r is replaced with $\hat{r}$, the process continues with $\hat{r}$, that is, another atom in the body of $\hat{r}$ is deleted and if the resulting rule is uniformly contained in $\hat{r}$, then $\hat{r}$ is replaced with that rule. The steps are summarized in the algorithm of Fig. 1. The final result is a rule which is unif[illegible]ly equivalent to the original one, but without any *redundant* atoms? i.e., atbms that can be deleted while preserving uniform equivalence. In proving the correctness of the algorithm, the only nontrivial point is to show that no atom has to be considered more than once. In other words, if some atom α is not redundant when it is considered for the first time, then subsequent deletions of other atoms cannot make α redundant. We shall formally prove this claim in the appendix Generally, the final result of the algorithm is not unique and may depend upon the order in which atoms are considered.

```
begin
repeat
    let $\alpha$ be an atom in the body of $r$ that has not yet been considered;
    let $\hat{r}$ be the rule obtained by deleting $\alpha$ from $r$;
    if $\hat{r} \subseteq^u r$ then replace $r$ with $\hat{r}$;
until each atom has been considered once;
end.
```

Fig. 1. Minimizing a rule r.

Example 8: Consider programs P_1 and P_2 of Example 7. Each one of these programs has a single rule, and the rule of P_2 is obtained from that of P_1 by deleting the atom $A(w, y)$. In Example 7, it is shown that $P_2 \subseteq^u P_1$. Thus, if we execute the algorithm of Fig. 1 with the rule of P_1 as input, then it is going to be replaced with the rule of P_2. It is easy to show that the rule of P_2 does not have any redundant atom. Therefore, the algorithm terminates with the rule of P_2 as the minimal form of the rule of P_1. •I

Redundant rules can be removed from a program P similarly to the elimination of redundant atoms from the body of a rule. A rule is deleted from P to obtain a program $\hat{P}$, and if $r \subseteq^u \hat{P}$, then $P \equiv^u \hat{P}$ and, so, $\hat{P}$ can replace P. In order to minimize a program P, we first minimize each rule by removing its redundant atoms, and then remove all redundant rules. However, the following situation is possible. An atom in some rule r of P may not be redundant if r alone is considered, but may be redundant if all the rules of P are considered. In other words, in order to minimize a rule r of P, we modify the algorithm of Fig. 1 by replacing the test $\hat{r} \subseteq^u r$ in the **if** statement with $\hat{r} \subseteq^u P$. The complete algorithm for minimizing a program P is given in Fig. 2. In the appendix, we prove that the final result of the algorithm has neither redundant rules nor redundant atoms. The only nontrivial part of the proof is showing that no rule or atom has to be considered more than once; the proof relies on the fact that at first each rule is minimized and only then redundant rules are removed. The final result of the algorithm is not necessarily unique.

VIII. Tuple-Generating Dependencies

A *tuple-generating* dependency (abbr. tgd) (Beeri and Vardi [1984], Fagin [1982], Yannakakis and Papadimitriou [1982]) is a formula of the form $\forall\bar{x}\exists\bar{y}[\psi_1(\bar{x}) \rightarrow \psi_2(\bar{x},\bar{y})]$, where $\bar{x}$ and $\bar{y}$ are vectors of variables and both ψ_1 and ψ_2 are conjunctions of atoms. We write a tgd without the quantifiers, e.g., $G(y, z) \rightarrow G(y, w) \wedge C(w)$ instead of $\forall y \forall z \exists w[G(y, z) \rightarrow G(y, w) \wedge C(w)]$. Universally quantified variables are those appearing in the left-hand side of the formula (these variables can also appear in the right-hand side). Existentially quantified variables are those appearing only in the right-hand side of the tgd. Note that the tgds considered in this paper are untyped.

```
begin
for each rule $r$ do
    repeat
        let $\alpha$ be an atom in the body of $r$ that has not yet been considered;
        let $\hat{r}$ be the rule obtained by deleting $\alpha$ from $r$;
        if $\hat{r} \subseteq^u P$ then replace $r$ with $\hat{r}$;
    until each atom has been considered once;
repeat
    let $r$ be a rule of $P$ that has not yet been considered;
    let $\hat{P}$ be the program obtained by deleting rule $r$ from $P$;
    if $r \subseteq^u \hat{P}$ then replace $P$ with $\hat{P}$;
until each rule has been considered once:
end.
```

Fig. 2. Minimizing a program *P*.

As usual, we say that a DB d satisfies a tgd τ if for every instantiation θ of the universally quantified variables, the following is true: If the left-hand side of τ is instantiated by θ to ground atoms of d, then the right-hand side of τ can also be instantiated to ground atoms of d by extending θ to an instantiation of all the variables of τ.

Example 9: Consider the tgd $G(x, y) \rightarrow A(y, z) \wedge A(z, x)$, and the DB produced in Example 2; recall that G is the transitive closure of A, and the DB is

$$\{A(1,2),\ A(1,4),\ A(4,1),\ G(1,2),\ G(1,4),\ G(4,1),\ G(1,1),\ G(4,4),\ G(4,2)\}$$

If we instantiate both x and y to 4, then the instantiated left-hand side, $G(4,4)$, is a ground atom of the DB. We can now choose to instantiate z to 1, and as a result, the instantiated right-hand side consists of ground atoms, $A(4,1)$ and $A(1,4)$, that are in the DB. The DB, however, does not satisfy the tgd, since instantiating x to 4 and y to 2 converts the left-hand side to a ground atom of the DB, but there is no possible instantiation of z that also converts the right-hand side to ground atoms of the DB. The tgd $G(x, y) \rightarrow G(x, z) \wedge A(z, y)$, on the other hand, is satisfied by the DB. For example, if x is instantiated to 1 and y to 2, then instantiating z to 1 converts the right-hand side to ground atoms of the DB. $\square$

Let S be a set of DBs. We say that program P_1 *uniformly* contains P_2 *over* S, written $P_2 \subseteq^u_S P_1$, if $P_2(d) \subseteq P_1(d)$ for all DBs $d \in$ S. In most cases we assume that S is the set of all DBs satisfying a given set T of tgds, and we usually denote this set by *SAT(T)*.

Tuple-generating dependencies are important in Datalog, because in many cases optimizing a program requires looking only at DBs that satisfy some tgds. One case is when the EDB satisfies some constraints that can be expressed as tgds. Using constraints in order to optimize programs has already been investigated (e.g., Chakravarthy et al. [1986]). We will show how to use tgds in a more general way. Essentially, we will give a proof procedure for showing $P_2 \subseteq^u_{SAT(T)} P_1$, and develop a technique for removing redundant atoms from a program P by doing the following steps. First, we have to show that $P_2 \subseteq^u_{SAT(T)} P_1$ for some suitable T, where P_2 is obtained from P_1 by deleting an atom α from some rule. Second, we have to show that $P_2 \subseteq^u_{SAT(T)} P_1$ implies $P_2 \subseteq P_1$. If we show both, then it follows that α is redundant in P_1, even if it is not redundant under uniform equivalence. This optimization technique will be described in detail later. In the remainder of this section, we will describe the first part of a procedure for determining whether $P_2 \subseteq^u_{SAT(T)} P_1$. The correctness of this procedure is proved in the appendix.

Considering Proposition 2, it comes as no surprise that in order to show $P_2 \subseteq^u_{SAT(T)} P_1$, we have to show $SAT(T) \cap M(P_1) \subseteq M(P_2)$, i.e., every model of P_1 that satisfies T is also a model of P_2. Moreover, the chase process can be easily modified to show that. As we shall see later, however, $SAT(T) \cap M(P_1) \subseteq M(P_2)$ alone does not imply $P_2 \subseteq^u_{SAT(T)} P_1$; and in the nest section we shall describe a second step that is needed in order to conclude that $P_2 \subseteq^u_{SAT(T)} P_1$.

In order to test whether $SAT(T) \cap M(P_1) \subseteq M(P_2)$, we have to consider each rule r of P_2 and show that when both P_1 and T are applied to the body of r, the result includes the head of r. Applying the tgds of T to a DB is similar to the application of rules, since tgds are also Horn clauses.

We will now describe how to apply the tgds in greater detail. There are two types of tgds: *full* tgds, namely, tgds without esistentially quantified variables, and embedded tgds, namely, tgds that have some existentially quantified variables. As illustrated by the following example, applying a full tgd to a DB is just the same as applying a rule.

Example 10: The tgd $A(x, y, z) \wedge B(w, y, v) \rightarrow A(x, y, v) \wedge T(w, y, z)$ is full. Applying it to a DB is the same as applying the following two rules. Note that each of these rules has the left-hand side of the tgd as its body, and one of the atoms in the right-hand side of the tgd as its head.

$A(x, y, v)$:- $A(x, y, z), B(w, y, v)$.
$T(w, y, z)$:- $A(x, y, z), B(w, y, v)$. □

An embedded tgd has existentially quantified variables and, therefore, in order to apply it we have to use Skolem functions. We follow the approach of database theory and view Skolem functions as nulls, i.e., unknown values. We denote nulls as $\delta_1, \ldots, \delta_i, \ldots$.

A tgd τ is applied to a DB as follows. Suppose that θ is an instantiation of the universally quantified variables of τ, such that θ shows that the DB violates τ. That is, θ converts the left-hand side of τ to ground atoms of the DB, and there is no extension of θ that also converts the right-hand side of τ to ground atoms of the DB. For each existentially quantified variable of τ, we choose a unique null δ_i (which is not already in the DB) and extend θ to an instantiation that maps each existentially quantified variable to its corresponding null. The instantiated atoms of the right-hand side of τ are added to the DB. For example, if τ is the tgd $G(x, y) \rightarrow A(x, w) \wedge G(w, y)$ and the atom $G(3, 2)$ is in the DB, then we add $A(3, \delta_{23})$ and $G(\delta_{23}, 2)$ (provided, of course, that the DB contains neither δ_{23} nor a pair of atoms of the form $A(3, e)$ and $G(e, 2)$, where e is either a constant or a null). The atoms $A(3, \delta_{23})$ and $G(\delta_{23}, 2)$ simply mean that there is some constant c such that $A(3, c)$ and $G(c, 3)$ are in the DB, but the actual value of c is unknown.

The combined application of a program P and a set of tgds T is denoted $[P, T]$. We apply $[P, T]$ to a DB d until no new atoms can be added to the **DB,** and the final result is denoted $[P, T](d)$. Clearly, $[P, T](d)$ is both a model of P and a DB that satisfies T. Since the application of tgds may add nulls that are not already in the DB, some sets of tgds can be applied to all initial DB forever. Note that once an atom with nulls is added to the DB, then it is viewed as a ground atom and nulls are viewed as constants, as far as applications of rules and tgds are concerned.

Example 11: Let P_1 be the program

$G(x, z)$:- $A(x, z)$.
$G(x, z)$:- $G(x, y)$, $G(y, z)$, $A(y, w)$.

and let P_2 be the program

$G(x, z)$:- $A(x, z)$.
$G(x, z)$:- $G(x, y)$, $G(y, z)$.

It. is easy to show that $P_1 \subseteq^u P_2$. We will show that $SAT(T) \cap M(P_1) \subseteq M(P_2)$, where T consists of the single dependency :

$G(x, z) \rightarrow A(x, w)$

The rules of P_2 have to be considered one by one; we start by instantiating the first rule of P_2 and, so, its body becomes the DB $\{A(x_0, z_0)\}$. Now we have to apply $[P_1, T]$ to this DB, and the result is $\{A(x_0, z_0), G(x_0, z_0)\}$ (note that only the first rule of P_1 can be applied to this DB). This result contains the instantiated head of the first rule of P_2.

Next, consider the DB $\{G(x_0, y_0), G(y_0, z_0)\}$, which is the instantiated body of the second rule of P_2. At first, the only possible application of $[P_1, T]$ is to apply the tgd of T. If the left-hand side of the tgd is instantiated to $G(y_0, z_0)$, then this instantiation cannot be extended to any instantiation that converts the right-hand side to a ground atom of the DB and, therefore, $A(y_0, \delta_1)$ is added to the DB. Similarly, the left-hand side of the tgd can be instantiated to $G(x_0, y_0)$, which results in adding $A(x_0, \delta_2)$ to the DB. Now, the body of the second rule of P_1 can be instantiated to $G(x_0, y_0)$, $G(y_0, z_0)$, $A(y_0, \delta_1)$, and so $G(x_0, z_0)$ is added to the DB, thereby showing that the instantiated head of the second rule of P_2 is in the result. Thus, we have shown that $SAT(T) \cap M(P_1) \subseteq M(P_2)$. In the next section, we will use this fact in order to conclude that $P_2 \subseteq^u_{SAT(T)} P_1$.

Showing $P_2 \subseteq^u_{SAT(T)} P_1$ is useful, because it implies $P_2 \subseteq P_1$ by the following simple argument (the argument is given here informally, and will be given formally in Section S). Applying program P_1 (or P_2) to an EDB, which is given as input, is the same as applying P_1 (or P_2) to the preliminary DB,‡ i.e., the DB consisting of the input and the ground atoms generated by the initialization rules (an initialization rule is a rule whose body has only extensional predicates). Since P_1 and P_2 have the same initialization rule, they have the same preliminary DB for every EDB; and it is easy to see that the preliminary DB satisfies T. Therefore, $P_2 \subseteq^u_{SAT(T)} P_1$ implies $P_2 \subseteq P_1$. Clearly, $P_1 \subseteq P_2$ and, so, $P_1 \equiv P_2$. It thus follows that the atom $A(y, w)$ in the second rule of P_1 is redundant under equivalence, although it is not redundant under uniform equivalence. □

‡ When P_1 (or P_2) is applied to the preliminary DB, the initialization rules are redundant and can be ignored.

IX. Preserving Tuple-Generating Dependencies

As stated earlier, SAT(T) $\cap$ $M(P_1) \subseteq M(P_2)$ alone does not imply $P_2 \subseteq^u_{SAT(T)} P_1$. As shown in the appendix, however, if we also show that P_1 *preserves* T, then $P_2 \subseteq^u_{SAT(T)} P_1$ follows. We say that P_1 preserves T if $P_1(d) \in SAT(T)$ for all DBs $d \in SAT(T)$.

It is not known whether there is a proof procedure for showing that a program P preserves a set of tgcls T. In this section we will describe a process that may efficiently show, in many practical cases, that P preserves T. The idea is to show that if we start with a DB $d \in SAT(T)$, then each iteration in the bottom-up computation of $P(d)$ preserves T. To espress the idea more formally, we need the following definitions. Applying P *non-recursively* to'a DB cl means applying it only to the ground atoms of d, and not to ground atoms generated from cl by previous applications. When P is applied non-recursively, we denote it as P^n. Clearly, the result of applying P^n to a DB d, denoted P"(d), is

$$\{ h\theta \mid \text{for some rule } h \text{ :- } b \text{ of } P \text{ and substitution } \theta, \text{ the atoms of } b\theta \text{ are in } d\}$$

Note that by our previous definitions, the output of $P(d)$ contains the input d. In comparison, P"(d) contains only the atoms generated by applying the rules non-recursively to d, but does not necessarily contain the atoms of d. This notation is just a matter of convenience, and should not cause any confusion.

Example 12: Let P be the program

$G(x, z)$:- $A(x, z)$.
$G(x, z)$:- $G(x,y)$, $G(y,z)$.

and let $d = \{A(1,2), G(2,3), G(3,4)\}$. P"(d) is $\{G(1,2), G(2,4)\}$, whereas $P(d)$ is $\{A(1,2), G(2,3), G(3,4), G(1,2), G(1,3), G(2,4), G(1,4)\}$. $\square$

Our idea is to show that P preserves T *non-recursively*, that is, $(d, P^n(d)\rangle \in SAT(T)$ for all $d \in SAT(T)$ (recall that $\langle d, P^n(d)\rangle$ is the union of d and P^n (cl)). Note that if P preserves T non-recursively, then P preserves T. The converse, however, is not necessarily true, that is, P may preserve T without preserving it non-recursively.

Proving that P preserves T non-recursively is done by a variant of the chase process that was originally proposed by Klug and Price [1982]. This process is complete for proving non-recursive preservation of T, that is, it terminates with a positive answer if indeed P preserves T non-recursively, but it may loop forever if T has embedded tgds and the answer is negative. Before fully defining this process, we illustrate it on a simple example.

Example 13: Consider the following recursive rule, denoted r,

$G(x,z)$:- $G(x,y)$, $G(y,z)$, $A(y,w)$.

and let τ be the tgd

`$G(x, z) \rightarrow A(x, w)$

In order to show that r preserves τ non-recursively, we will attempt to prove the opposite by trying to construct a counterexample, and if we fail to do so, then r preserves τ non-recursively. A counterexample, in this particular case, is a DB $d \in SAT(\tau)$ such that

$\langle d, r^n(d)\rangle$ violates τ. The DB $\langle d, r^n(d)\rangle$ violates τ if it has a ground atom $G(x_0, z_0)$ that *exhibits* a violation of τ, that is, a ground atom $G(x_0, z_0)$ such that for all w_0, the DB $\langle d, r^n(d)\rangle$ does not have a ground atom of the form $A(x_0, w_0)$. A ground atom $G(x_0, z_0)$ of $\langle d, r^n(d)\rangle$ that exhibits a. violation of τ must be in $r^n(d)$ (it cannot be in d, since $d \in SAT(\tau)$). Therefore, we will try to build a counteresaniple by first assuming that $G(x_0, z_0)$ is in $r^n(d)$, and then adding atoms to d that are needed in order to
(1) have the atom $G(x_0, z_0)$ in $r^n(d)$, and
(3) make d satisfy τ.

The atom $G(x_0, z_0)$ can be in $r^n(d)$ only as a result of applying r^n to d. By unifying $G(x_0, z_0)$ with the head of r, we can determine which ground atoms must be in d in order to produce $G(x_0, z_0)$. In this particular case, the unification shows that d must have the following atoms:

$$G(x_0, y_0),\ G(y_0, z_0),\ A(y_0, w_0)$$

where y_0 and w_0 are some constants.

Since d satisfies τ, it is possible to apply the tgd τ to d. Applying τ to $G(x_0, y_0)$ yields $A(x_0, \delta_1)$, and applying it to $G(y_0, z_0)$ yields $A(y_0, \delta_2)$. Note that these applications result in ground atoms that must be in d (as opposed to applications of r^n that produce ground atoms in P(d)). Basically, the applications of τ correspond to inferences implied by the fact that d satisfies τ and by the fact that certain ground atoms are known to be in d. In principle, the tgcl τ should be applied repeatedly to the atoms of d (both the atoms that have originally been in d and those added to d by previous applications of the tgcl). In this particular case, the tgcl can be applied only to the ground atoms originally known to be in d (i.e., those produced by unifying $G(x_0, z_0)$ with the head of r). Consequently, the ground atoms that must be in d ase

$$G(x_0, y_0), G(y_0, z_0), A(y_0, w_0), A(x_0, \delta_1), A(y_0, \delta_2).$$

Among them there is $A(x_0, \delta_1)$, which shows that $G(x_0, z_0)$ does not exhibit a violation of τ. Therefore, there is no counterexample $\langle d, r^n(d)\rangle$ and, so, r preserves τ non-recursively. $\square$

We can now generalize the above esample to an arbitrary *P* and *T*. In order to prove that *P* preserves *T* non-recursively, we do the following for each $\tau \in T$. First, the left-hand side of τ is instantiated by replacing each variable with a distinct constant. The ground atoms of the instantiated left-hand side are treated according to one of the following two cases:
(1) Ground atoms of extensional predicates become part of d.
(2) Ground atoms of intentional predicates become part of $P^n(d)$.
For each ground atom α in $P^n(d)$, we should add to d some atoms that produce α when *P* is applied non-recursively to d. In general, there are many ways to add atoms that produce α. Each possible way is determined by some rule with a head that can be unified with α. Thus, we should consider all possible combinations of unifying the ground atoms that have been added to *P"(d)* with heads of rules (if there are n ground atoms in *P"(d)* and each can be unified with m rules, then there are nm combination to consider). Essentially, we have to show that for each possible combination, there is no violation of τ. So, consider one possible combination that unifies each ground atom α of an intentional predicate G

(in the instantiated left-hand side of τ) with the head of some rule r for G. As a result of the unification, the variables of r that appear in the head are instantiated to constants, In order to convert the the body of r to ground atoms, the rest of the variables of r are instantiated to new distinct constants, and the ground atoms of the body are added to d. In summary, d contains all the ground atoms that are either

(1) atoms of extensional predicates from the instantiated left-hand side of the tgd τ, or

(2) atoms (extensional or intentional) from bodies of rules that have been unified with atoms of intentional predicates from the left-hand side of τ.

As for $P^n(d)$, it contains atoms of intentiona. predicates from the instantiated left-hand side of τ.

In the second step, the tgds of T (all of them - not just τ) are applied to d to produce more ground atoms that must be in d. The tgds are applied repeatedly, until no more ground atoms can be generated from existing ones (and, consequently, d becomes a DB that satisfies T).

In the third step,† the program P is applied non-recursively to d to get $P^n(d)$.

In the final step, we should check whether $\langle d, P^n(d)\rangle$ satisfies τ. In order to check that, it is sufficient to consider the instantiated left-hand side of τ (which is part of $P^n(d)$), and check whether it exhibits a violation of τ in $\langle d, P^n(d)\rangle$. No violation is exhibited if the instantiation of the left-hand side can be extended to an instantiation that also includes the existentially quantified variables‡ of τ, such that the right-hand side of τ becomes a subset of $\langle d, P^n(d)\rangle$. In fact, it follows that there is no need to compute all of $P^n(d)$; instead, it is sufficient to determine whether $\langle d, P^n(d)\rangle$ contains ground atoms showing that the instantiated left-hand side of τ does not exhibit a violation. For clarity of presentation, however, we will continue to use the step that computes $P^n(d)$.

The program P preserves T non-recursively, if for all $\tau \in T$ and for all combinations of unifying the instantiated left-hand side of τ with rules' heads of P, no violation of τ is exhibited.

In Esample 13, the left-hand side of the tgd τ has only one atom and there is only one rule; therefore, there is only one combination to check, and as has been shown, it does not eshibit a violation. The steps for checking whether P preserves T non-recursively are summarized in Fig. 3. Finer details of the algorithm are explained in the nest two paragraphs.

The step of applying T to d may not terminate if new nulls are repeatedly introduced. It is still possible, however, to terminate the inner loop in finite time (for any particular choice of $\tau \in T$ and any particular choice of rules for τ) if no violation of τ is exhibited. In order to achieve that, the last three steps of the inner loop should be interleaved as follows. First, T is applied to d to produce some more new atoms that must be in d. Next, $P^n(d)$ is computed again, since its value may have changed as a result of the new atoms that have just been added to d. The third step is to check whether the instantiated left-hand side of τ exhibits a violation in the current $\langle d, P^n(d)\rangle$. If no violation is exhibited, then τ is preserved and there is no need to continue. If a violation still exists, then the previous

† In Esample 13, this step is redundant and, hence, has been omitted.

‡ Recall that the existentially quantified variables of a tgd are those appearing only in the right-hand side.

```
begin
repeat
    make $d$ empty;
    choose a $\tau \in T$;
    let $\theta$ map the universally quantified variables of $\tau$ to distinct constants;
    instantiate the left-hand side of $\tau$ according to $\theta$;
    add the instantiated atoms of extensional predicates to $d$;
    repeat
        choose a rule for each instantiated atom of an intentional predicate;
        unify each atom with the head of the rule chosen for it,
            and add the instantiated body to $d$;
        apply the tgds of $T$ to $d$;
        compute $P^n(d)$;
        check whether the instantiated left-hand side
            exhibits a violation of $\tau$ in $\langle d, P^n(d)\rangle$;
    until a violation has been exhibited or
            all combinations of choosing rules have been examined;
until a violation has been exhibited or all $\tau \in T$ have been chosen;
if a violation has been exhibited
    then $P$ does not preserve $T$ non-recursively
    else $P$ preserves $T$ non-recursively
end.
```

Fig. 3. Procedure for testing non-recursive preservation of T.

steps should be reiterated.

As already said, each atom of an intentional predicate, in the instantiated left-hand side of τ, is unified with the head of some rule. This has the effect of testing whether τ is satisfied when atoms of intentional predicates in its left-hand side are restricted to be in $P^n(d)$. In Esample 13, there is a single atom in the left-hand side of τ, and therefore, τ is satisfied in $\langle d, P^n(d)\rangle$ if the following is shown:
(1) τ is satisfied in $\langle d, P^n(d)\rangle$ when the left-hand side is restricted to be in $P^n(d)$, and
(2) τ is satisfied in $\langle d, P^n(d)\rangle$ when the left-hand side is restricted to be in d.
Part (1) has been shown in Example 13 by unifying the left-hand side with the head of r. Part (2) follows immediately from the fact that d satisfies τ. The situation, however, is not that simple if the left-hand side of τ has more than one atom of an intentional predicate. In this case, we have to check that τ is also satisfied when some atoms† are in $P^n(d)$, while others are in d. Thus, we should consider more combinations than stated earlier. The combinations are all those in which an atom of an intentional predicate in the left-hand side of τ is either unified with the head of some rule or is assumed to be in d (without, of course, being unified with any rule). If an atom is unified with the head of some rule,

† The atoms referred to are, of course, the instantiated atoms of intentional predicates in the left-hand side of τ.

then the atom becomes part of $P^n(d)$, while the instantiated body of the rule becomes part of cl. We can still stick to the old definition of the combinations to be considered if for each intentional predicate Q, we add a trivial rule of the form: $Q(x_1, \ldots, x_n)$:- $Q(x_1, \ldots, x_n)$. From now on we will assume that each program is augmented with these trivial rules (although usually we do not explicitly write these rules as part of the program). Therefore, the combinations to be considered are the same as defined originally, that is, a. combination unifies each atom of an intentional predicate with the head of some rule.

Example 14: Consider again the program P_1 given in Example 11:

$G(x, 3)$:- $A(x, z)$.
$G(x, z)$:- $G(x, y), G(y, z), A(y, w)$.

and the tgcl τ:

$$G(x, z) \rightarrow A(x, w)$$

We will show that P_1 preserves $T = \{\tau\}$ non-recursively, and hence it also preserves T. Combining this with the fact $SAT(T) \cap M(P_1) \subseteq M(P_2)$, which was shown in Example 11, implies that $P_2 \subseteq^u_{SAT(T)} P_1$. Let $G(x_0, z_0)$ be the instantiated left-hand side of τ. In Example 13, we have shown that no violation is exhibited when $G(x_0, z_0)$ is unified with the head of the second rule of P_1. Similarly, there can be no violation when $G(x_0, z_0)$ is unified with the trivial rule $G(x, z)$:- $G(x, z)$.‡ The last case to consider is unifying with the rule:

G(x, 3) :- $A(x, z)$.

As a result of unifying $G(x_0, z_0)$ with the head of the above rule, d becomes the DB $\{A(x_0, z_0)\}$. The tgds of T cannot be applied to d. Nest, by applying P_1^n to d, we get that $P_1^n(d)$ is $\{G(x_0,z_0)\}$. Since $A(x_0, z_0)$ is in $\langle d, P_1^n(d)\rangle$, no violation of τ is exhibited and, therefore, P_1 preserves T, as was claimed. □

Example 15: Let r be the same rule as in Example 13, that is

$G(x, z)$:- $G(x, y), G(y, z), A(y, w)$.

and let the tgd τ be

$$G(x, y) \wedge G(y, z) \rightarrow A(y, w)$$

We will show that r (i.e., the program consisting of r) preserves τ non-recursively. Recall that we should treat the program as if it also has the trivial rule

$G(x, z)$:- $G(x, z)$.

and, hence, there are four possible combinations of unifying the atoms in the left-hand side of τ with rules' heads. So let

$G(x_0, y_0), G(y_0, z_0)$

‡ As a general rule, there can be no violation if the left-hand side of the tgd has only one atom of an intentional predicate and the unification is done with a trivial rule, because the whole instantiated left-hand side becomes part of d, which is assumed to satisfy T. The trivial rules have to be used only when we deal with a tgd that, has more than one atom of an intentional predicate in its left-hand side (see Example 15).

be the instantiated left-hand side of τ, and consider the following four combinations.

Combination 1. $G(x_0, y_0)$ is unified with the head of r and, as a result, the following ground atoms (i.e., those from the body of r) are in d:

$$G(x_0, y_1),\ G(y_1, y_0),\ A(y_1, w_0)$$

and $G(y_0, z_0)$ is unified with the head of the trivial rule, which adds the following atom to d:

$$G(y_0, z_0)$$

Now $T = \{\tau\}$ should be applied to d and, actually, only the following application is possible. The left-hand side of τ is instantiated to the following ground atoms of cl:

$$G(y_1, y_0), G(y_0, z_0)$$

Since this instantiation cannot be extended to one that also converts the right-hand side to ground atoms of d, the ground atom $A(y_0, \delta_1)$ is added to d. Note that no more applications of T are possible after this one. The atom $A(y_0, \delta_1)$ of cl shows that no violation of τ is exhibited in (d, $r^n(d)\rangle$ for the combination being considered.

Combination 2. $G(x_0, y_0)$ is unified with the head of the trivial rule and, as a result, the following ground atom is added to d:

$$G(x_0, y_0)$$

and $G(y_0, z_0)$ is unified with the head of r, and the following ground atoms are added to d:

$$G(y_0, y_1),\ G(y_1, z_0),\ A(y_1, w_0)$$

Now $T = \{\tau\}$ is applied to d. Again, there is only possible application, which is obtained by instantiating the left-hand side of τ to the following ground atoms of d:

$$G(x_0, y_0),\ G(y_0, y_1)$$

This instantiation adds $A(y_0, \delta_1)$ to d, and this ground atom shows that no violation of τ is eshibited in (d, $r^n(d)\rangle$.

Combination 3. $G(x_0, y_0)$ is unified with the head of r, and the following ground atoms are added to d:

$$G(x_0, y_1),\ G(y_1, y_0),\ A(y_1, w_0)$$

and $G(y_0, z_0)$ is also unified with the head of r, and the following ground atoms are added to d:

$$G(y_0, y_2),\ G(y_2, z_0),\ A(y_2, w_1)$$

Now $T = \{\tau\}$ is applied to d by instantiating the left-hand side of τ to the following ground atoms of d:

$$G(y_1, y_0),\ G(y_0, y_2)$$

and, as result of this instantiation, $A(y_0, \delta_1)$ is added to d. The atom $A(y_0, \delta_1)$ shows that no violation of τ is exhibited in (d, $r^n(d))$ for the combination being considered.

Combination 4. Both $G(x_0, y_0)$ and $G(y_0, z_0)$ are unified with the head of the trivial rule and, therefore, become part of d. Clearly, there cannot be a violation in this case, since d satisfies T.

Since no combination exhibits a violation, r preserves τ. □

Example 16: Consider the rule r

$$G(x, z) \text{ :- } A(x, y), G(y, z), G(y, w), C(w).$$

and the following tgd, denoted τ

$$G(y, z) \rightarrow G(y, w) \wedge C(w)$$

To show that r preserves τ non-recursively, we instantiate the left-hand side of τ to

$$G(y_0, z_0)$$

and unify it with the head of r. Consequently, the following ground atoms are in d:

$$A(y_0, y_1),\ G(y_1, z_0),\ G(y_1, w_0),\ C(w_0)$$

Note that in this case, the tgd τ cannot be applied to d to produce new atoms. But when r is applied non-recursively to d, the DB $r^n(d)$ becomes equal to

$$G(y_0, z_0),\ G(y_0, w_0)$$

To see that $G(y_0, z_0)$ is in $r^n(d)$, note that this atom was unified with the head of r and the instantiated body became a part of d. To see that $G(y_0, w_0)$ is in r^n (cl), instantiate the variables of r as follows. Instantiate x to y_0, y to y_1, and both z and w to w_0.

The ground atoms $G(y_0, w_0)$ and $C(w_0)$ show that $\langle d, r^n(d)\rangle$ does not violate τ when the left-hand side is instantiated to $G(y_0, z_0)$. Thus, r preserves τ. □

X. Determining Equivalence

In this section we will show how it is sometimes possible to infer that $P_2 \subseteq P_1$ from the fact that $P_2 \subseteq^u_{SAT(T)} P_1$. Later we will discuss how to use this technique in order to optimize programs. But first we need some definitions. A rule r of a program P is an initialization rule if the body of r has only extensional predicates. P^i is the program consisting of the initialization rules of P. Note that P^i is a non-recursive program. Given an EDB d as an input for P, we define $P^i(d)$ to be the set of ground atoms generated by applying P^i to d (since P^i is a non-recursive program, $P^i(d)$ is defined in the same way as applying a program non-recursively, i.e., $P^i(d)$ does not include d). The *preliminary* DB for an EDB d is $\langle d, P^i(d)\rangle$.

Example 17: Let P be the program

$$G(x, z) \text{ :- } A(x, z).$$
$$G(x, z) \text{ :- } G(x, y), G(y, z).$$

and let $d = \{A(1,2), A(2,3), A(3,4)\}$. $P^i(d)$ is $\{G(1,2), G(2,3), G(3,4)\}$, and the preliminary DB for d is $\{A(1,2), A(2,3), A(3,4), G(1,2), G(2,3), G(3,4)\}$. □

In the nest example, we illustrate how to infer $P_2 \subseteq P_1$ from $P_2 \subseteq^u_{SAT(T)} P_1$.

Example 18: Consider again the two programs of Example 11. Recall that P_1 is the program

$G(x, z)$:- $A(x, z)$.
$G(x, z)$:- $G(x, y)$, $G(y, z)$, $A(y, w)$.

and P_2 is the program

$G(x, z)$:- $A(x, z)$.
$G(x, z)$:- $G(x, y)$, $G(y, z)$.

Clearly, $P_1 \subseteq^u P_2$. In Example 11 we have shown that SAT(T) $\cap$ $M(P_1) \subseteq M(P_2)$, where T consists of the single tgd

$G(x, z) \rightarrow A(x, w)$

and in Example 14 we have shown that P_1 preserves T. Consequently, $P_2 \subseteq^u_{SAT(T)} P_1$. In this example, we will show that $P_2 \subseteq P_1$ and hence $P_1 \equiv P_2$, since $P_1 \subseteq^u P_2$ (and so $P_1 \subseteq P_2$).

First., we will show that for every EDB d, the preliminary DB, $\langle d, P_1^i(d)\rangle$, satisfies T. Recall that P_1^i consists of the rule

$G(x, z)$:- $A(x, z)$.

Essentially, the procedure of Fig. 3 is used to show that $\langle d, P_1^i(d)\rangle$ satisfies T. There are, however, two important changes. First, we do not assume that cl satisfies T and, therefore, we omit the step in which the tgds of T are applied to cl. Second, d is an EDB given as an input to P_1 and, so, it does not have any ground atom of an intentional predicate. Therefore, we do not add to the program P_1^i the trivial rules (i.e., rules of the form $Q(x_1, \ldots, x_n)$:- $Q(x_1, \ldots, x_n)$) for the intentional predicates.

Thus, we start by instantiating the left-hand side of the only tgd in T, and the result, is $G(x_0, z_0)$. There is only one rule in P_1^i and, hence, only one combination of unifying the instantiated left-hand side with heads of rules. This unification results in d being the DB $\{A(x_0, z_0)\}$. Since $A(x_0, z_0)$ has just been shown to be in $\langle d, P_1^i(d)\rangle$, no violation of the tgd is exhibited and, therefore, the preliminary DB of P_1 satisfies T.

P_1 and P_2 have the same initialization rule and, consequently, their preliminary DBs are the same (when given the same EDB as input). Therefore $P_2 \subseteq^u_{SAT(T)} P_1$ implies $P_2 \subseteq P_1$, since the preliminary DB satisfies T. $\square$

To sum up the approach illustrated in the above example, showing $P_2 \subseteq P_1$ entails showing the following:

(1) SAT(T) $\cap$ $M(P_1) \subseteq M(P_2)$.
(2) P_1 preserves T.
(3) For all EDBs cl, programs P_1 and P_2 have the same preliminary DB.
(4) The preliminary DB always satisfies T.

Part (1) can be shown using the chase process described in Section VIII. Part (2) is shown using the process summarized in Fig. 3. Part (3) requires showing that P_1^i and P_2^i are equivalent. Equivalence of non-recursive programs is the same as uniform equivalence and, thus, there is an algorithm for showing that (i.e., the one described in Section VI). In fact, equivalence of non-recursive programs is the same as equivalence of unions of tableaus (Sagiv and Yannakakis [1979]). Part (4) can be shown by the procedure of Fig. 3 with the following modificat ions. First, the step of applying the tgcls of T to d is removed. Second, the program (i.e, P_1^t) is not augmented with trivial rules for the intentional predicates.

The above recipe for showing $P_2 \subseteq P_1$ has some drawbacks that may limit its applicability. First, it is not always clear how to find a set of tgds T for which (1)–(4) hold. Moreover, the fact that $P_2 \subseteq P_1$ does not necessarily imply that there is such a T. Second, the procedure for testing (1) (or (2)) terminates in finite time if the answer is positive, but may loop forever if the answer is negative. Nevertheless, we believe that in many practical cases this approach is useful in optimizing programs.

We end this section with an important comment on conditions (1)–(4) above. Actually, it is not necessary to consider the preliminary DBs of both P_1 and P_2. Instead, it is sufficient to consider only the preliminary DB of P_1 and show that it satisfies T. In other words, conditions (3) and (4) can be replaced with the following condition:

(3') The preliminary DB of P_1 satisfies T.

The reason for that is as follows. We know that $P_2 \subseteq^u_{SAT(T)} P_1$ and we want to conclude that $P_2 \subseteq P_1$, that is, we want to show that if d is an EDB, then $P_2(d) \subseteq P_1(d)$. So, let d' be a preliminary DB of P_1 obtained from d, i.e., $d \subset d'$; and suppose that d' satisfies T. Since $P_2 \subseteq^u_{SAT(T)} P_1$ and d' satisfies T, it follows that

$$P_2(d') \subseteq P_1(d') \qquad (A)$$

But Datalog programs are monotonic and, therefore,

$$P_2(d) \subseteq P_2(d') \qquad (B)$$

because $d \subset d'$. Moreover, d' is a preliminary DB of P_1, i.e., it is obtained by applying some rules of P_1 to d and, hence,

$$P_1(d) = P_1(d') \qquad (C)$$

From (A), (B), and (C) it follows that

$$P_2(d) \subseteq P_1(d)$$

When defining the preliminary DB, it is not necessary to choose the one generated by the initialization rules. Instead, it is sufficient to consider any set of rules of P_1 and apply it a fixed number of times to the initial EDB given as an input. Applying a given set of rules a fised number of times (even if the rules are recursive) can be expressed in terms of non-recursive rules and, hence, testing whether the preliminary DB satisfies T can be clone as described earlier (i.e., as described for a preliminary DB created by the initialization rules).

XI. Optimizing Under Equivalence

In Example 18, we have shown that the atom $A(y, w)$ is redundant in the recursive rule of P_1. Note that this cannot be shown using the algorithm of Fig. 2, because $A(y, w)$ is not redundant under uniform equivalence. Compared to optimization under uniform equivalence, it is less clear how to carry out this type of optimization algorithmically. The problem is how to find a. tgd that shows the redundancy of $A(y, w)$. In practice, the appropriate approach is to use some heuristics. In trying to generalize Example 18, note that the tgd used in that example, i.e., $G(x, z) \rightarrow A(x, w)$, has the property that the following can be shown very easily.

$$SAT(T) \cap M(P_1) \subseteq M(P_2) \tag{1}$$

Recall that T consists of the above tgd and P_2 is obtained from P_1 by removing $A(y, w)$ (see Example 18 for more details). More specifically, a single application of T to the body of the recursive rule of P_2 makes that body identical to the body of the recursive rule of P_1 and, in effect, shows (1).

The above idea for choosing a tgd can be phrased in terms of the following syntactical properties. In order to make the following properties as clear as possible, recall that the rule that has been optimized in Example 18 is

$$G(x, z) \text{ :- } G(x, y), G(y, z), A(y, w).$$

and the chosen tgcl can also be written as $G(y, z) \rightarrow A(y, w)$, i.e., it consists of atoms appearing in the body of the above rule and having the following properties.

(1) The left-hand side of the tgd has the same predicate as the head of the rule being optimized.

(2) If the tgd has a variable w that appears only in its right-hand side, then all the atoms (from the rule's body) that contain w are in the right-hand side of the tgd.

(3) All the variables of the tgd that appear only in its right-hand side are not in the rule's head.

Once a tgd has been chosen, the next step is to test whether the atoms in the right-hand side of the tgd are redundant in the rule's body.

It is not difficult to devise heuristics that look for a tgd satisfying the above properties. Once a tgd is found, it remains to check the conditions specified in the previous section. -This is just a matter of syntactical manipulation, which is conceptually easy. The only 'problem is that it may not terminate. The common way of handling an optimization process that may run too long is to spend on optimization a predetermined amount of time. As a last example, we illustrate the above ideas.

Example 19: Consider the following program.

$$G(x, z) \text{ :- } A(x, z), C(z).$$
$$G(x, z) \text{ :- } A(x, y), G(y, z), G(y, w), C(w).$$

Clearly, a candidate tgd for showing redundancy is

$$G(y, z) \rightarrow G(y, w) \wedge C(w)$$

which will be denoted by τ. Let P_1 be the original program. and let P_2 be the one obtained by deleting $A(y, w)$ and $C(w)$ from the body of the recursive rule of P_1. Clearly, $P_1 \subseteq^u P_2$. We will show that $P_2 \subseteq P_1$ by showing the following:
(1) $SAT(T) \cap M(P_1) \subseteq M(P_2)$.
(2) P_1 preserves τ.
(3') The preliminary DB of P_1 satisfies T.
It is easy to show that (1) holds. In Example 16, it was shown that the recursive rule of P_1 preserves T. Since T has a single tgd with only one atom in its left-hand side, (3') and the fact that the recursive rule of P_1 preserves T imply (2). Thus, it only remains to show that (3') holds. So let $G(y_0, z_0)$ be the instantiated left-hand side of the tgd τ. Unifying it with the head of the rule of P_1^i produces the DB $\{A(y_0, z_0), C(z_0)\}$. The ground atoms $G(y_0, z_0)$ and $C(z_0)$ show that there is no violation of τ, when its left-hand side is instantiated to $G(y_0, z_0)$. Thus. the preliminary DB satisfies τ. We can, therefore, conclude that the atoms $A(y, w)$ and $C(w)$ are redundant in the recursive rule of P_1. $\square$

XII. Conclusion and Open Problems

We have given an algorithm for minimizing Datalog programs under uniform equivalence. This minimization reduces the number of joins needed to find all the answers to a query. We have also given an algorithm for testing uniform containment (and hence also uniform equivalence) of programs, which may be useful when other types of optimizations are considered. The results on uniform containment and minimization can be extended to Datalog programs with stratified negation, and in a forthcoming paper, we will describe how it is clone.

We have considered the problem of testing uniform containment when the DB satisfies some constraints that are expressed as tuple-generating dependencies. $P_2 \subseteq^u_{SAT(T)} P_1$ is implied by the following two conditions:
(1) $SAT(T) \cap M(P_1) \subseteq M(P_2)$.
(2) P_1 preserves T.
Condition (1) can be tested by the chase process of Section VII, which always terminates with the correct answer if there are only full tgds. If there are also embedded tgds, then the chase may not terminate when (1) is not true. As for condition (2), the procedure of Section VIII can prove it in some, but not all, cases in which it is true. That procedure may not terminate if there are embedded tgcls. There is, however, an important case in which (2) is obviously true, namely, when the tgds have only extensional predicates on the left-hand side (condition (2) is true in this case, because the evaluation of P_1 never adds new ground atoms of extensional predicates). In particular, if the tgds espress constraints that the EDB satisfies, then they have only extensional predicates; and in this case, the chase process (for testing condition (1)) can be used to transform a program to an equivalent one that may be more efficient, as done, for example, by Chakravarthy et al. [1986].

We have also shown how to use the procedures for determining (1) and (2) in order to optimize programs under equivalence. Some heuristics are needed to carry out this type of optimization, but we believe that this optimization technique can be applied easily and usefully in practice.

Some open problems remain. First, it is important to characterize cases in which the the procedures for testing (1) and (2) are guaranteed to terminate. It is easy to give ad-hoc generalizations based on examples shown in this paper. However, is it possible to find some nontrivial cases:)

Another important open problem is to characterize cases that have algorithms foi finding tgds that show redundancy whenever some atoms are redundant, or at least, whenever recluntlancy can be shown by some tgds. If no algorithms can be found, then more heuristics should be developed for finding tgds that may show redundancy.

Acknowledgments

The author thanks Jeff Finger, Georg Gottlob, Paris Kanellakis, Michael Kifer, Dave Maier, Oclecl Shmueli, Jeff Ullman, and Allen Van Gelder for helpful comments and discussions.

References

A. V. Aho, Y. Sagiv, and J. D. Ullman [1979]. "Equivalences among relational expressions," *SIAM J.* Computing, **8:2**, pp. X8-246.

F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman [1986a]. "Magic sets and other strange ways to Implement Logic Programs," *Proc. Fifth ACM SIGACT-SIGMOD Symp. on Principles of Database Systems,* March 24-26, Cambridge, Mass., pp. 1-15.

F. Bancilhon and R. Ramakrishnan [1986b]. "An amateur's introduction to recursive query processing strategies," *Proc. ACM SIGMOD ht. Conf. on Management of Data,* May 28-30, Washington, D.C., pp. 16–52.

C. Beeri, A. 0. Mendelzon, Y. Sagiv, and J. D. Ullman [1981]. "Equivalence of relationa. database schemes?" *SIAM J.* Computing. **10:2**, pp. 352–370.

C. Beeri and M. Y. Varcli [1984]. "A proof procedure for data dependencies," *J. A CM.* **31:4**, pp.718–741.

U. S. Chakravarthy, J. Minker, and J. Grant [1986]. "Semantic Query Optimization: Additional Constraints and Control Strategies," *Proc. First Int. Conf. on Expert Database Systems,* April 1-4, Charleston, S. C., pp. 259–269.

A. Ii. Chanclra and P. M. Merlin [1976]. "Implementation of conjunctive queries in relational databases," *Proc. Ninth ACM SIGACT Symp.* on Theory of Computing, May, pp. 77–90.

S. S. Cosmadakis and P. C. Kanellakis [1986]. "Parallel evaluation of recursive rule queries," *Proc. Fifth ACM SIGACT-SIGMOD Symp. on Principles* of *Database Systems.* March 24-26, Cambridge, Mass., pp. 280–293.

R. Fagin [1982]. "Horn clauses and database dependencies," *J. ACM,* **29:4**, pp. 952–983.

.J. J. Finger [1986]. "Exploiting constraints in design synthesis," Ph.D. Thesis, (in preparation), Department of Computer Science, Stanford University, Stanford, CA 94305.

H. Gaifman [1986]. Private communication.

H. Gallaire and J. Minker (eds.) [1978]. *Logic and Databases*, Plenum, New York.

L. J. Henschen and S. A. Naqvi [1984]. "On compiling queries in recursive first-order databases," *J. ACM,* **31:1**, pp. 47–85.

M. Iiifer and E. L. Lozinskii [1986]. "Filtering data flow in deductive databases," *Proc. Int. Conf. on Database Theory,* Sept. S-10, Rome, Italy.

J. J. King [1981]. "Query optimization by semantic reasoning," Ph.D. Thesis, (also Rept. No. STAN-CS-81-857), Department of Computer Science, Stanford University, Stanford, CA 94305.

A. Klug and R.. Price [1982]. "Determining view dependencies using tableaux," *ACM Trans. on Database Systems* **7**:3, pp. 361-380

E. L. Lozinskii [1985]. "Evaluating queries in deductive databases by generating," *Proc. 9th IJCAI*, pp.173–177.

D. Maier, A. 0. Mendelzon, and Y. Sagiv [1979]. "Testing Implications of Data Dependencies," *ACM Trans. on Database Systems* **4**:4, pp. 455-469

D. McKay and S. Shapiro [1981]. "Using active connection graphs for reasoning with recursive rules ," *Proc. 7th IJCAI*, pp. 368-374.

J. Rohmer and R.. Lescoeur [1985]. "The Alexander Method: A Technique for the Processing of Recursive Axioms in Deductive Databases," Bull Internal Report.

D. Saccà and C. Zaniolo [1986]. "On the implementation of a simple class of logic queries for dat abases ," *Proc. Fifth ACM SIGACT-SIGMOD Symp. on Principles of Database Systems,* March 24-26, Cambridge, Mass., pp. 16-23.

Y. Sagiv, and M. Yannakakis [1980]. "Equivalences among relational expressions with the union and difference operators," *J. ACM,* **27:4**, pp. 633-655.

0. Shmueli [1986]. "Decidability and expressiveness aspects of logic queries," unpublished manuscript, Computer Science Dept., Technion - Israel Institute of Technology, Haifa 32000, Israel.

J. D . Ullman [1985]. "Implementation of logical query languages for databases," *ACM Trans. on* Database *Systems* **10**:3, pp. 289-321

M. H. Van Emden and R. A. Iiowalski [1976]. "The semantics of predicate logic as a programming language," *J.* ACM, **23:4**, pp. 733-742.

A. Van Gelder [1986]. "A message passing framework for logical query evaluation," *Proc. ACM SIGMOD Int. Conf. on Management of Data,* May 28-30, Washington, D.C., pp. 155-165.

M. Yannakakis and C. H. Papadimitriou [1982]. "Algebraic dependencies," *J. Comput. Syst. Sci.,* **25**, pp. 2-41.

Appendix

I. Correctness of Testing Uniform Containment

We first prove two lemmas about the relationship between uniform containment of two programs and containment of their sets of models. Proposition 2, which is stated in Section IV, follows as a. special case of these lemmas. Similar lemmas, but for a very restricted class of rules, were proved by Beeri et al. [1981].

Let S denote a set of DBs. Note that S can be any set, and not necessarily the set of DBs that satisfy some set T of tgds. For a program P, the set $P(S)$ consists of all outputs for inputs in S, that is, $P(S) = \{P(d) \mid d \in S\}$. Recall that $M(P)$ is the set of all models of P.

Lemma 1: $P_2 \subseteq_S^u P_1 \Longrightarrow S \cap M(P_1) \subseteq M(P_2)$.

Proof: Suppose that $P_2 \subseteq_S^u P_1$. Let $d \in S \cap M(P_1)$. We claim that the following is true:

$$d \subseteq P,(d) \subseteq P_1(d) = d \tag{1}$$

In proof, the left containment holds, because the output of every program contains its input. The right Containment holds, since $P_2 \subseteq_S^u P_1$ and $d \in$ S. The equality holds, because $d \in M(P_1)$. Therefore, (1) implies that $d \in M(P_2)$. □

Lemma 2: $P_2 \subseteq_S^u P_1 \Longleftarrow P,(S) \cap M(P_1) \subseteq M(P_2)$.

Proof: Suppose that $P,(S) \cap M(P_1) \subseteq M(P_2)$, and let $d \in$ S be an input for P_2 (d is not necessarily a model of P_2). Let $d_1 = P_1(d)$ and $d_2 = P_2(d)$. We have to show that $d_2 \subseteq d_1$. Since $d_1 \in P_1(S) \cap M(P_1)$, it follows that $d_1 \in M(P_2)$. Therefore, $d_2 \subseteq d_1$, since d_2 is the minimal model of P_2 that contains d and, clearly, d_1 is also a model of P_2 that contains d. •I

The previous two lemmas imply the following corollary. Proposition 2 is a special case of this corollary when S is the set of all DBs.

Corollary 1: Let P_1 be a program and S a set of DBs such that $P_1(S) \subseteq$ S. Then $P_2 \subseteq_S^u P_1 \Longleftrightarrow S \cap M(P_1) \subseteq M(P_2)$. □

Note that if S is the set of all DB satisfying the tgds of some T, i.e., S = $SAT(T)$, then P_1 (S) $\subseteq$ S means that P_1 preserves T.

Clearly, $SAT(T) \cap M(P_1) \subseteq M(P_2)$ if and only if $SAT(T) \cap M(P_1) \subseteq M(r)$ for all rules r of P_2, because a DB is a model of P_2 if and only if it is a model of r for all rules r of P_2. The chase process, described in Section VIII, tests $SAT(T) \cap M(P) \subseteq M(r)$, and the following theorem proves its correctness. Recall that in order to perform this process, the body of r has to be viewed as a DB, and this is accomplished by instantiating the variables of r to distinct constants according to some substitution θ. Also recall that the combined application of a program P and a set of tgds T, which is explained in Section VIII, is denoted by $[P, T]$.

Theorem 1: Let r be the rule h :- b, i.e., h is the head and b is the body, and let θ be a one-to-one mapping of the variables of r to constants that do not already appear in r. Then

$$h\theta \in [P,T](b\theta) \Longleftrightarrow SAT(T) \cap M(P) \subseteq M(r)$$

Proof: The main idea of the proof is the same as in Maier et al. [1979]. First, we assume that *SAT(T)* $\cap$ $M(\ P\ ) \subseteq M(\ r\ )$ and will show that $h\theta \in [P,$ T] $(b\theta)$. So, consider the DBs $b\theta$ and *[P, T](b*θ). Clearly, *[P,* $T](b\theta) \in$ *SAT(T)* $\cap$ $M(P)$, since *[P,* $T](b\theta)$ is defined to be the DB obtained from $b\theta$ by applying the rules of *P* and tgcls of *T* until no rule or tgcl can be applied anymore. Therefore, *[P , T] (be)* $\in M(\ r\ )$, because we have assumed *SAT(T)* $\cap$ *M(P)* $\subseteq M(\ r)$.

We will now show that [P, $T](b\theta) \in M(\ r\ )$ implies $h\theta \in$ *[P, T](be),* which is what we have to prove. By definition, the DB $[P,T](b\theta)$ contains *be.* If we apply r to $b\theta$, it is immediately clear that $h\theta \in r(b\theta)$, because when the body of *r* is instantiated according to θ, it becomes $b\theta$ and, therefore, $h\theta$ is in the output. But $[P,T](b\theta)$ is assumed to be a model of *r* and, so, applying *r* to *[P,* $T](b\theta)$ cannot generate any new atom. Therefore, $h\theta \in [P,T](b\theta)$, because applying *r* to $b\theta$, which is contained in $[P,\ T]($ *be),* produces $h\theta$.

We will now prove the other direction, namely, we assume that $h\theta\ \in$ *[P, T](be).* and will show that *SAT(T)* $\cap$ $M(P) \subseteq M(r)$. *So,* let *d* be any DB in *SAT(T)* $\cap$ $M(P)$. We have to show that $d \in$ *M(r).* To show that, we consider an arbitrary substitution ρ that instantiates the body of *r* (i.e., *b)* to ground atoms of d. Now, to complete the proof, we have to show that *hp* is also in *d.* But $h\theta \in$ *[P, T](* $b\theta$) and, so, there is a sequence of substitutions $\varphi_1, \ldots, \varphi_n$ that shows $h\theta \in [P,T](b\theta)$, that is, for each i there is either a rule of *P* or a tgd of *T,* such that when the rule or tgd is instantiated according to φ_i, a new atom is generated, and the last application (i.e., the one for φ_n) generates $h\theta$. Thus, it follows that ρ o θ^{-1} o $\varphi_1, \ldots .\ \rho$ o θ^{-1} o φ_n is a sequence of instantiations that shows that . $[P,T](d)$ contains *hp.* But $d \in$ *SAT(T)* $\cap$ *M(P)* implies *d = [P,T](d)* and, so, *hp* is in cl. cl

Note that if *T* has embedded tgds, then the DB *[P,* $T](b\theta)$ may be infinite† and, therefore, there is no bound on the time it may take to clisover that *[P,* $T](b\theta)$ contains $h\theta$ (although $h\theta$ will be discovered within a finite time if it is indeed in *[P, T] (be)).* Moreover, if $[P,T](b\theta)$ does not contain $h\theta$, then it may be impossible to determine this fact just by computing *[P,* $T](b\theta)$, since the computation may be infinite. Also note that if *[P,* $T](b\theta)$ does not contain $h\theta$, then it could be that the only DBs d, such that $d \in$ *SAT(T)* $\cap$ *M(P)* and $d \notin$ *M(r),* are infinite. In other words, if *T* includes embedded tgds, then the direction $\Longleftarrow$ in Theorem 1 is true provided that the set of all possible DBs include both infinite and finite DBs. Clearly, if there are no tgds at all, then we have the following important corollary of Theorem 1, which is the proof of correctness for the algorithm for testing uniform containment that is given in Section VI. This algorithm always terminates.

Corollary 2: Let *r* be a rule with head *h* and body *b,* and let θ be a one-to-one mapping of the variables of *r* to constants that do not already appear in *r.* Then

$$h\theta \in P(b\theta) \Longleftrightarrow \textit{i\%!!(P)} \subseteq M(r)$$

II. Correctness of Testing Non-Recursive Preservation of Tgds

The procedure of Section IX for testing whether a program *P* preserves non-recursively a set *T* of tgds is also based on the chase. It is similar to the one described by Klug

† This happens when repeated applications of embedded tgds create ground atoms with new nulls.

and Price [1982], and we shall not prove it formally here. Suffices to say that if the procedure either determines that P does not preserve non-recursively some $\tau \in T$ or does not terminate, then it actually constructs a DB d such that d satisfies T and (cl, $P^n(d)\rangle$ violates τ. Note that the procedure may not terminate only if T has embedded tgds, and in this case the counterexample d is infinite. If the procedure determines that P preserves T non-recursively, then it essentially does that by constructing for each potential violation of some $\tau \in T$, a canonical DB in which that violation does not exist, and that canonical DB can be mapped homomorphically into any other DB that might exhibit the same violation. Therefore, no violation is possible.

III. Correctness of the Algorithm for Minimizing Programs

Theorem 2: The algorithm of Section VII for minimizing programs under uniform equivalence is correct.

Proof: Essentially, we have to show that no atom or rule has to be considered more than once. So, let P_f be the final program produced by the algorithm. We have to show that P_f has neither redundant rules nor redundant atoms. We will first show that P_f does not have any redundant rule.

Suppose that some rule r is redundant in P_f . Let P denotes the program at the beginning of the iteration in which rule r was considered for deletion; and let $\hat{P}$ and $\hat{P}_f$ denote programs P and P_f , respectively, with r removed. Clearly, P and P_f are uniformly equivalent, since the algorithm deletes while preserving uniform equivalence. Since r has not been deleted permanently, $r \not\subseteq^u \hat{P}$. Let h and b be the head and body, respectively, of rule r, and let θ be a one-to-one mapping of the variables of r to constants not already in r. We have $h\theta \notin \hat{P}(b\theta)$, since $r \not\subseteq^u \hat{P}$, and we also have $h\theta \in \hat{P}_f(b\theta)$, since r is redundant in P_f. But this is a contradiction to $\hat{P}_f(b\theta) \subseteq \hat{P}(b\theta)$, which follows from the fact that every rule of $\hat{P}_f$ is also a rule of $\hat{P}$ (note that here we have used the fact that redundant atoms are deleted before redundant rules and, therefore, a rule that appears in P_f has exactly the same body also in P; if some rule had appeared in P_f with some atoms deleted from its body, as compared to P, it would have been impossible to infer $\hat{P}_f(b\theta) \subseteq \hat{P}(b\theta)$). Thus, we have shown that P_f does not have any redundant rule.

Now suppose that some rule r of P_f (i.e., the final program) has a redundant atom α in its body. P denotes the program at the beginning of the iterations in which α was considered for deletion. Let h be the head of r, and let b and b_f be its bodies in P and P_f , respectively (note that every atdm of b_f is also in b). The bodies $\hat{b}_f$ and $\hat{b}$ are obtained from b_f and b, respectively, by deleting a. Let θ be a one-to-one mapping of all the variables of r to constants not already in r. Since α has not been deleted permanently, $h\theta \notin P(k)$. Since α is redundant in P_f, it follows that $h\theta \in P_f(\hat{b}_f\theta)$. But this is a contradiction, since $P_f \equiv^u P$ and, therefore, $P_f(\hat{b}_f\theta) \subseteq P(\hat{b}\theta)$, because $\hat{b}_f\theta \subseteq \hat{b}\theta$, and Datalog programs are monotone, that is, adding more atoms to the input does not remove any atom from the output. Thus, we have also shown that P_f does not have any redundant atom. □