A BASIS FOR DEDUCTIVE DATABASE SYSTEMS

J. W. LLOYD AND R. W. TOPOR

This paper provides a theoretical basis for deductive database systems. A deductive database consists of closed typed first order logic formulas of the form $A \leftarrow W$, where $A$ is an atom and $W$ is a typed first order formula. A typed first order formula can be used as a query, and a closed typed first order formula can be used as an integrity constraint. Functions are allowed to appear in formulas. Such a deductive database system can be implemented using a PROLOG system. The main results are the soundness of the query evaluation process, the soundness of the implementation of integrity constraints, and a simplification theorem for implementing integrity constraints. A short list of open problems is also presented.

1. INTRODUCTION

In recent years, there has been a growing interest in deductive database systems [4–7, 15]. Such systems have first order logic as their theoretical foundation. This approach has several desirable properties. Logic itself has a well-understood semantics. Furthermore, its use as a foundation for database systems means that we can employ logic as a uniform language for data, programs, queries, views, and integrity constraints.

One of the most promising approaches to implementing deductive database systems is to use a PROLOG system as the query evaluator [2, 8, 10, 12, 17, 18]. This approach requires some restrictions on the kinds of formulas which can be used in the database. However, such deductive databases are substantially more general than relational databases and can still be implemented efficiently.

Address correspondence to Dr. J. W. Lloyd, Department of Computer Science, University of Melbourne, Parkville, Victoria 3052, Australia.

THE JOURNAL OF LOGIC PROGRAMMING
52 Vanderbilt Ave., New York, NY 10017
0743-1066/85/$03.30
This paper contains some basic theoretical results for such an approach to deductive database systems. In particular, it builds on earlier work in [10], which contains special cases of some of the results presented here. In [10], to simplify matters, we assumed that there were no functions in databases, integrity constraints, or queries. In this paper that restriction is removed. It turns out that the proof of a key lemma (Lemma 1 below) is considerably more difficult when functions are allowed.

The major results of this paper are the soundness of query evaluation and the soundness of the implementation of integrity constraints. These results give a firm theoretical foundation in a general setting for the approach of implementing deductive database systems using PROLOG. Also presented is a simplification theorem for implementing integrity constraints which extends a similar result for relational databases given in [13].

In Section 2, we introduce the main concepts used in these results. In Section 3, the soundness of the query evaluation process is proved. In Section 4, we prove that the implementation of integrity constraints is sound and we prove the simplification theorem. The last section contains some open problems.

We assume familiarity with [10] and also the basic theoretical results of logic programming, which can be found in [9]. The notation and terminology of this paper is consistent with [9] and [10].

2. BASIC CONCEPTS

In this section, we introduce the concepts of a deductive database, query, and integrity constraint. We also give the definition of the completion of a database and a correct answer substitution.

We emphasize that, in contrast to [10], here we allow functions to appear in databases, queries, and integrity constraints. The introduction of functions does cause certain problems (see [14] for a discussion), and hence they are commonly excluded in the database context. The major reason for excluding functions is that they can cause the set of answers to a query to be infinite and hence affect the ability of the system to return all answers. However, as we show, having functions does not affect soundness in any way and, after all, soundness is the prime theoretical requirement of any database system. In any case, at this stage, it is important to push the theoretical developments as far as possible.

Underlying all the theoretical developments is a typed first order language. We assume that the language contains only finitely many constants, functions, and predicates. Each predicate, function, constant, and variable is typed. Predicates have type denoted $\tau_1 \times \cdots \times \tau_n$, and functions have type denoted $\tau_1 \times \cdots \times \tau_n \to \tau$. If $f$ has type $\tau_1 \times \cdots \times \tau_n \to \tau$, we say $f$ has range type $\tau$. Terms in the language have a type induced in the obvious way. We assume that, for each type $\tau$, there is a ground term of type $\tau$. We use the notation $\forall x/\tau W$ and $\exists x/\tau W$ to indicate that the bound variable $x$ of the quantifier is of type $\tau$. $\forall(F)$ denotes the typed universal closure of the formula $F$. We also use $\forall$ to denote the ordinary type-free universal closure. It will always be clear from the context which is meant. The concepts of interpretation, model, logical consequence, and so on, are defined in the natural way for typed first order logic (also called many-sorted first order logic). Background material on types is contained in [3].
The reason for using a typed language is evident. Types provide a natural way of expressing the domain concept of relational databases. The requirement that formulas be correctly typed ensures that important kinds of integrity constraints are maintained.

Next we turn to the definitions of the main concepts. For examples of these concepts, see [lo].

**Definition.** A *database clause* is a typed first order formula of the form

\[ A \leftarrow W \]

where \( A \) is an atom and \( W \) is a typed first order formula. \( A \) is called the *head* and \( W \) the *body* of the clause. The formula \( W \) may be absent. Any variables in \( A \) and any free variables in \( W \) are assumed to be universally quantified at the front of the clause.

**Definition.** A *database* is a finite set of database clauses.

**Definition.** A *query* is a typed first order formula of the form

\[ \leftarrow W \]

where \( W \) is a typed first order formula and any free variables of \( W \) are assumed to be universally quantified at the front of the query.

**Definition.** Let \( \leftarrow W \) be a query, where \( W \) has free variables \( x_1, \ldots, x_n \). An *answer substitution* is a substitution for some or all of the variables \( x_1, \ldots, x_n \).

It is understood that substitutions are correctly typed in that each variable is bound to a term of the same type as the variable.

As in [lo], our soundness results require the introduction of the completion of a database. The definition of the completion given here is a generalization of the definition given in [lo]. This generalization is needed because we are now allowing functions to appear in formulas. The definition of the completion requires the introduction of a typed equality predicate \( =_\tau \) for each type \( \tau \). These predicates are assumed not to appear in the original language. In particular, no database, query or integrity constraint contains any \( =_\tau \).

**Definition.** Let \( D \) be a database and \( p \) a predicate occurring in \( D \). Suppose the predicate \( p \) has definition

\[ A_1 \leftarrow W_1 \\
\vdots \\
A_k \leftarrow W_k, \]

where each \( A_i \) has the form \( p(t_1, \ldots, t_n) \). Then the *completed definition* of \( p \) is the formula

\[ \forall x_1/\tau_1 \cdots \forall x_n/\tau_n (p(x_1, \ldots, x_n) \leftrightarrow E_1 \lor \cdots \lor E_k), \]

where \( x_1, \ldots, x_n \) are variables not appearing in any \( A_i \leftarrow W_j \), each \( E_i \) has the form

\[ \exists y_1/\sigma_1 \cdots \exists y_d/\sigma_d ((x_1 =_{\tau_1} t_1) \land \cdots \land (x_n =_{\tau_n} t_n) \land W_j), \]
and $y_1, \ldots, y_d$ are the variables of $A_i \leftarrow W_i$ which are universally quantified at the front of the clause.

**Definition.** Let $D$ be a database and $p$ a predicate occurring in $D$. Suppose there is no clause in $D$ with predicate $p$ in its head. Then the **completed definition** of $p$ is the formula

$$\forall x_1/\tau_1 \cdots \forall x_n/\tau_n \neg p(x_1, \ldots, x_n).$$

The equality theory for a database consists of all axioms of the following form:

1. $c \neq_d d$, where $c$ and $d$ are distinct constants of type $\tau$.
2. $\forall (f(x_1, \ldots, x_n) \neq \gamma g(y_1, \ldots, y_m))$, where $f$ and $g$ are distinct functions of range type $\tau$.
3. $\forall (f(x_1, \ldots, x_n) \neq \gamma c)$, where $c$ is a constant of type $\tau$ and $f$ is a function of range type $\tau$.
4. $\forall (t[x] \neq \gamma x)$, where $t[x]$ is a term of type $\tau$ containing $x$ and different from $x$.
5. $\forall ((x_1 \neq \gamma_1 y_1) \vee \cdots \vee (x_n \neq \gamma_n y_n)) \rightarrow f(x_1, \ldots, x_n) \neq \gamma f(y_1, \ldots, y_n)$, where $f$ is a function of type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$.
6. $\forall x/\tau (x = \gamma x)$.
7. $\forall ((x_1 = \gamma_1 y_1) \wedge \cdots \wedge (x_n = \gamma_n y_n)) \rightarrow f(x_1, \ldots, x_n) = \gamma f(y_1, \ldots, y_n)$, where $f$ is a function of type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$.
8. $\forall ((x_1 = \gamma_1 y_1) \wedge \cdots \wedge (x_n = \gamma_n y_n)) \rightarrow (p(x_1, \ldots, x_n) \rightarrow p(y_1, \ldots, y_n))$, where $p$ (including every $=$) is a predicate of type $\tau_1 \times \cdots \times \tau_n$.
9. $\forall x/\tau ((x = \gamma a_1) \vee \cdots \vee (x = \gamma a_k) \vee (\exists x_1/\tau_1 \cdots \exists x_n/\tau_n (x = \gamma f_1(x_1, \ldots, x_n)) \vee \cdots \vee (\exists y_1/a_1 \cdots \exists y_m/a_m (x = \gamma f_m(y_1, \ldots, y_m))))$, where $a_1, \ldots, a_k$ are all the constants of type $\tau$ and $f_1, \ldots, f_n$ are all the functions of range type $\tau$.  

Axioms 1 to 8 are the typed versions of the usual equality axioms for a program [9]. The axioms 9 are the **domain closure axioms**. This equality theory generalizes the equality theory given in [10] for the function-free case.

**Definition.** Let $D$ be a database. The completion of $D$, denoted $\text{comp}(D)$, is the collection of completed definitions for each predicate in $D$ together with the above equality theory.

**Definition.** Let $D$ be a database and $Q$ a query $\leftarrow W$. A **correct answer substitution** for $\text{comp}(D) \cup \{Q\}$ is an answer substitution $\theta$ such that $\forall (W \theta)$ is a logical consequence of $\text{comp}(D)$.

The concept of a correct answer substitution gives a declarative understanding of the desired output from a query to a database. In the next section, we prove the soundness of an implementation of this concept.

Next we turn to integrity constraints.

**Definition.** An **integrity constraint** is a closed typed first order formula.

Intuitively, an integrity constraint should be an invariant of the database. This leads us to make the following definition.
Definition [15]. Let \( D \) be a database such that \( \text{comp}(D) \) is consistent, and let \( W \) be an integrity constraint. We say \( D \) satisfies \( W \) if \( W \) is a logical consequence of \( \text{comp}(D) \); otherwise, we say \( D \) violates \( W \).

Finally we define a class of databases that has several important properties.

Definition. A database is called hierarchical if its predicates can be partitioned into levels so that the definitions of level 0 predicates consist solely of database clauses \( A \leftarrow \) and the bodies of the clauses in the definitions of level \( j \) predicates (\( j > 0 \)) contain only level \( i \) predicates, where \( i < j \).

Such a database is more general than a relational database, but does not allow recursively defined predicates. Related definitions are given in [1] and [16].

3. QUERIES

In this section, we shall prove that our query evaluation process is sound. To prove this result, we only have to prove a generalization of Lemma 4 of [10] for which functions are allowed. The remainder of the argument given in [10] is valid in this more general context. The generalization of Lemma 4 of [10] which we require is given by Lemma 1 below.

The precise details of the query evaluation process are given in [10]. Fortunately, most of the details are not needed to understand Lemma 1. Thus we only present here an overview of query evaluation. The first step of the query evaluation process transforms typed first order formulas into corresponding type-free first order formulas. For this, we use a standard transformation [3].

Definition. Let \( W \) be a typed first order formula. For each type \( \tau \), we associate a unary type predicate also denoted by \( \tau \). Then the type-free form \( W^* \) of \( W \) is the first order formula obtained from \( W \) by applying the following transformations to subformulas of \( W \) of the form \( \forall x/\tau \forall V \) and \( \exists x/\tau V \):

(a) Replace \( \forall x/\tau V \) by \( \forall x(V \leftarrow \tau(x)) \).

(b) Replace \( \exists x/\tau V \) by \( \exists x(V \land \tau(x)) \).

We will also require the usual type theory [3].

Definition. The type theory \( \Phi \) consists of all axioms of the following form:

(1) \( \tau(a) \), where \( a \) is a constant of type \( \tau \).

(2) \( \forall x_1 \cdots \forall x_n(\tau(f(x_1, \ldots, x_n)) \leftarrow \tau_1(x_1) \land \cdots \land \tau_n(x_n)) \), where \( f \) is a function of type \( \tau_1 \times \cdots \times \tau_n \rightarrow \tau \).

Now we can give an overview of query evaluation. To answer a query \( Q \) to a database \( D \), we first transform \( Q \) and \( D \) to their type-free forms \( Q^* \) and \( D^* \), where \( D^* = \{ C^* : C \in D \} \). We then transform \( Q^* \) and \( D^* \cup \Phi \) into an ordinary PROLOG goal \( G \) and program \( P \) (which generally may include negations) by successively applying some of the 10 transformation rules given in [10], which eliminate universal
quantifiers, implications, and so on, in the bodies of clauses. A computed answer to the query \( Q \) for the database \( D \) is then defined to be a computed answer to the goal \( G \) for the program \( P \). Note that, due to the presence of the type predicates, every computed answer is a ground substitution for all free variables in the body of the query. As we explained in [10], to ensure that the negations are handled properly, it is essential that the PROLOG system use a safe computation rule (that is, one which only selects negative literals that are ground). If \( R \) is a safe computation rule, then an \( R \)-computed answer substitution for \( D \cup \{Q\} \) is an \( R \)-computed answer substitution for \( P \cup \{G\} \).

Since we are allowing functions, a query can have infinitely many answers. However, under a reasonable restriction on the type theory \( \Phi \), we can ensure that each query can have at most finitely many answers. As with databases, we say that \( \Phi \) is hierarchical if there are some types whose type axioms are only of the form (1) above (that is, these types do not have any function of that range type), there are some further types whose axioms of the form (2) above can only refer to the first set of types in their bodies, and so on. In particular, this restriction bans recursion in \( \Phi \). For such a type theory, it is clear that there are only finitely many ground terms of each type. Consequently, each query can have at most finitely many answers. We emphasize that it is not so much the presence of functions which causes queries to have infinitely many answers, but rather the presence of a "recursive" type theory.

With this background, we now proceed with the proof of Lemma 1. The lemma is a technical one which is only concerned with the first step of query evaluation, where we transform typed formulas into type-free ones. In this lemma, \( D^* \cup \Phi \) is essentially a type-free database (called an extended program in [10]), and its completion, \( \text{comp}(D^* \cup \Phi) \), is essentially a type-free version of the completion of a database given above, without the domain closure axioms. We refer the reader to [10] for the precise definitions.

Lemma 1. Let \( D \) be a database and \( W \) a closed typed first order formula. Let \( D^* \) and \( W^* \) be the type-free forms of \( D \) and \( W \). If \( W^* \) is a logical consequence of \( \text{comp}(D^* \cup \Phi) \), then \( W \) is a logical consequence of \( \text{comp}(D) \).

Proof. The proof is rather long and requires some preparation. Given a model \( M \) for \( \text{comp}(D) \), we have to construct a model \( M^* \) for \( \text{comp}(D^* \cup \Phi) \). The complexity of the construction of \( M^* \) which we use is needed to ensure that the equality axioms are satisfied.

Let \( M \) be a model for \( \text{comp}(D) \). Using (the typed version of) [11, p. 83], we can assume without loss of generality that \( M \) is normal, that is, each \( =_r \) is assigned the identity relation on the domain \( C_r \) of type \( \tau \). We can also suppose the \( C_r \)'s are disjoint. Put \( C = \bigcup_r C_r \).

The underlying language \( L^* \) for the interpretation \( M^* \) includes all the constants, functions, and (nonequality) predicates of the underlying language \( L \) for \( M \). \( L^* \) differs from \( L \) in that all type information is suppressed, the various typed equality predicates \( =_\tau \) are replaced by a single equality predicate \( =_r \), and there is a unary predicate \( \tau \) for each type \( \tau \).

Let \( F^* \) be the set of mappings on the \( C_r \) assigned by \( M \) to the functions in \( L \). Let \( T \) be the set of all (free) terms that can be formed using elements of \( C \) as primitive terms and elements of \( F^* \) as functions. (Note that the type restrictions are ignored in
forming these terms). The domain of $M^*$ will be the set of equivalence classes of a particular equivalence relation $\Delta$ on $T$.

To define $\Delta$, we introduce a reduction operation on $T$. We write $f'(d_1, \ldots, d_n) \rightarrow d$ if $f$ has type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$, $f'$ is the mapping assigned to $f$ by $M$, $d_i \in C_{\tau_i}$, $d \in C_\tau$, and $f'(d_1, \ldots, d_n) = d$. For $s, t \in T$, we write $s \Rightarrow t$ if $t$ is the result of replacing some (not necessarily proper) subterm $f'(d_1, \ldots, d_n)$ of $s$ by $d$, where $f'(d_1, \ldots, d_n) \rightarrow d$. We say that $s \in T$ is irreducible if there is no $t \in T$ such that $s \Rightarrow t$. Finally, for $s, t \in T$, we say that $s$ reduces to $t$ if there exists $r_0, r_1, \ldots, r_n \in T$ such that $s = r_0 \Rightarrow r_1 \Rightarrow \cdots \Rightarrow r_n = t$.

Now we can define the equivalence relation $\Delta$ on $T$. Let $s, t \in T$. Then $s \Delta t$ if there exists $u \in T$ such that $s \Rightarrow u$ and $t \Rightarrow u$. To prove that $\Delta$ is an equivalence relation, we use the following lemma.

**Lemma 2.** Let $s \in T$. Then there exists a unique irreducible $t \in T$ such that $s$ reduces to $t$. (We say that $t$ is the irreducible form of $s$.)

**Proof of Lemma 2.** That there exists an irreducible form of each $s \in T$ is immediate, since in each reduction $u \Rightarrow v$, $v$ has fewer subterms than $u$.

To prove that irreducible forms are unique, first note that if $f'(s_1, \ldots, s_n)$ reduces to $g'(t_1, \ldots, t_m)$, then $f' = g'$, and that the last step in any reduction of $f'(s_1, \ldots, s_n)$ to an element $d \in C$ hence has the form $f'(d_1, \ldots, d_n) \rightarrow d$. Structural induction can then be used to show that the assumption that $s$ has two distinct irreducible forms leads to a contradiction.

**Lemma 3.** $\Delta$ is an equivalence relation.

**Proof of Lemma 3.** Clearly, $\Delta$ is reflexive and symmetric. That $\Delta$ is transitive follows immediately from Lemma 2.

We now define the domain of the model $M^*$ to be $T/\Delta$, the set of $\Delta$-equivalence classes in $T$. If $t \in T$, we let $[t]$ denote the $\Delta$-equivalence class containing $t$. Note that $T/\Delta$ contains a copy of $C$ via the injective mapping $d \mapsto [d]$. Thus, in essence, we have simply enlarged $C$ in a particular way to obtain a domain for $M^*$.

If $c$ is a constant in $L^*$ and $M$ assigns $c' \in C$ to $c$, then $M^*$ assigns $[c']$ in $T/\Delta$ to $c$. Let $f \in L^*$ be an $n$-ary function. Suppose $M$ assigns the mapping $f'$ to $f$. Then $M^*$ assigns the mapping from $(T/\Delta)^n$ into $T/\Delta$ defined by $([t_1], \ldots, [t_n]) \mapsto [f'(t_1, \ldots, t_n)]$ to $f$. It is easy to see that this mapping is well defined. Note that this mapping is an extension of $f'$.

Suppose $p$ is an $n$-ary predicate in $L^*$. If $M$ assigns the relation $p'$ to $p$, then $M^*$ assigns the relation $\{(d_1, \ldots, [d_n]) : (d_1, \ldots, d_n) \in p'\}$ on $(T/\Delta)^n$ to $p$. To a type predicate $\tau$, $M^*$ assigns the unary relation $\{[d] : d \in C_\tau\}$. In essence, $M^*$ assigns $C_\tau$ to $\tau$. Finally, $M^*$ assigns the identity relation on $T/\Delta$ to $\equiv$.

This completes the definition of the interpretation $M^*$ for $\text{comp}(D^* \cup \Phi)$. We now check that $M^*$ is a model for $\text{comp}(D^* \cup \Phi)$. Much of the verification is routine, and we take the liberty of omitting some details.

We first check that $M^*$ is a model for the equality theory of $\text{comp}(D^* \cup \Phi)$. The eight axioms of the equality theory are given in [10] or [9, p. 70]. Apart from axiom
(4), these axioms are easily seen to be satisfied. Axiom (4) is

$$\forall (t[x] \neq x), \text{ where } t[x] \text{ is a term containing } x \text{ and different from } x.$$ 

That this axiom is satisfied follows immediately from the next lemma.

**Lemma 4.** Let $r, s \in T$. If $r$ is a proper subterm of $s$, then $r \not\Delta s$.

**Proof of Lemma 4.** Suppose $r \Delta s$. Then there exists an irreducible $t \in T$ such that $r$ reduces to $t$ and $s$ reduces to $t$. Let $u \in T$ be the result of replacing the occurrence of $r$ in $s$ by $t$. Then $t$ is a proper subterm of $u$, and $u$ reduces to $t$. If $t \in C$, then we obtain a contradiction using axiom (4) of the equality theory for $D$. Otherwise, $t$ has the form $f'(t_1, \ldots, t_n)$, in which case we again have a contradiction, since it is impossible for $u$ to reduce to $t$. □

The remainder of the verification that $M^*$ is a model for $\text{comp}(D^* \cup \Phi)$ depends on another lemma. For this we need a definition. A *variable assignment* $V$ wrt $M$ is an assignment to each variable $x$ in $L$ of an element $d \in C_\tau$, where $\tau$ is the type of $x$. Corresponding to $V$, there is a variable assignment $V^*$ wrt $M^*$ which assigns $[d]$ to $x$.

**Lemma 5.** Let $W$ be a (not necessarily closed) typed first order formula, $V$ a variable assignment wrt $M$, and $V^*$ the corresponding variable assignment wrt $M^*$. Then $W$ is true wrt $M$ and $V$ iff $W^*$ is true wrt $M^*$ and $V^*$.

This lemma is a variant of a well-known result (Lemma 43A in [3]). The proof is a straightforward induction argument on the structure of $W$.

Using Lemma 5, it can now be checked that $M^*$ is a model for the remainder of $\text{comp}(D^* \cup \Phi)$. The domain closure axioms for $D$ are used to show that $M^*$ is a model for the only-if halves of the completed definitions of the type predicates.

We have now finally shown that $M^*$ is a model for $\text{comp}(D^* \cup \Phi)$. Since $W^*$ is a logical consequence of $\text{comp}(D^* \cup \Phi)$, we have that $M^*$ is a model for $W^*$. Using Lemma 5 again, we obtain that $M$ is a model for $W$. Thus $W$ is a logical consequence of $\text{comp}(D)$. This completes the proof of Lemma 1. □

We can use Lemma 1 in place of Lemma 4 of [10] to obtain the following theorem, which is a generalization of Theorem 3 of [10].

**Theorem 1.** Let $D$ be a database, $Q$ a query, and $R$ a safe computation rule. Then every $R$-computed answer substitution for $D \cup \{Q\}$ is a correct answer substitution for $\text{comp}(D) \cup \{Q\}$.

Theorem 1 is the fundamental result which guarantees the soundness of our query evaluation process. The proof of this theorem (which includes Lemma 1 and several lemmas and theorems in [10]) is indeed long and complicated. However, it would be a mistake to conclude that the implementation of our query evaluation process is correspondingly complicated. In fact, the opposite is the case. The main part of the implementation concerns the 10 transformations given in [10]. These can be implemented in a PROLOG program which contains one clause for each transformation.
4. INTEGRITY CONSTRAINTS

In this section, we prove that our implementation of integrity constraints is sound. We also prove that our simplification method for implementing integrity constraints is sound.

The standard method for determining whether a database satisfies or violates an integrity constraint $W$ is by evaluating the query $\leftarrow W$. The idea is as follows. We evaluate the query $\leftarrow W$. If this query succeeds (that is, if we obtain an SLDNF-refutation), then Theorem 2 below shows that $D$ satisfies $W$. Similarly, if the query fails finitely (that is, if we obtain a finitely failed SLDNF-tree), then Theorem 2 shows that $D$ violates $W$. For the precise definitions of these concepts, we refer the reader to [10]. Theorem 2 below generalizes Theorems 4 and 5 of [10]. The proof is exactly as in [10], except the Lemma 1 above is used instead of Lemma 4 of [10].

Theorem 2. Let $D$ be a database, $W$ an integrity constraint, and $R$ a safe computation rule. Suppose $\text{comp}(D)$ is consistent.

(a) If there exists an SLDNF-refutation of $D \cup \{ \leftarrow W \}$ via $R$, then $D$ satisfies $W$.

(b) If $D \cup \{ \leftarrow W \}$ has a finitely failed SLDNF-tree via $R$, then $D$ violates $W$.

Next we turn to the simplification method for implementing integrity constraints. Let $D$ be a database. Suppose a user requests that some fact $A$ be deleted from $D$. Since $D$ is a deductive database, $A$ may not be explicitly present in $D$, but instead be a logical consequence of $D$. Thus, to perform the user’s request, the system may instead delete some other fact (or facts) explicitly present in the database. This will result in $A$ no longer being a logical consequence of $D$. Intuitively, we expect the deleted fact (or facts) to be “minimal”, that is, their deletion should change $D$ as little as possible. In relational database terminology, finding the right fact (or facts) to delete is called the view update problem. For an addition to a deductive database the situation is much simpler, since we can explicitly add the fact.

In fact, we are not directly concerned with this problem here. We assume that, for whatever reason, the system has to either add a clause to a database or delete an (explicitly present) clause from the database. Such an update can cause an integrity constraint to be violated. The simplification method is concerned with the problem of checking with the least amount of work that all the integrity constraints are still satisfied. The key idea is to use the fact that an integrity constraint was satisfied before the update was made either to eliminate the integrity constraint from further consideration or to construct simplified versions of it which must then be checked. The intention is that the simplified versions will be easier to check than the original constraint. This idea is well known in the context of relational databases (see [13] and the references therein). We prove that this simplification method is also sound for deductive databases. In this context, matters are greatly complicated by the presence of rules.
To cover the most general situation with a single theorem, we use the concept of a
transaction. A transaction is a finite sequence of additions of clauses to a database
and deletions of clauses from a database. If \( D \) is a database and \( t \) is a transaction,
then the application of \( t \) to \( D \) produces a new database \( D' \), which is obtained by
applying in turn each of the deletions and additions in \( t \). We assume that, in any
transaction, we do not have the addition and deletion of the same clause. As the
deletions and additions in a transaction can then be performed in any order, we
assume that all the deletions are performed before the additions. With regard to
integrity constraint checking, a transaction is indivisible, so we need only check the
constraints at the end of the transaction. Note that we can use a single transaction to
pass from any database \( D \) to any other database \( D' \).

The results which follow all concern databases, which, by definition, are based on
a typed language. The proofs of these results use various definitions and results from
[9]. In fact, we will actually require the typed versions of these definitions and
results. In all cases, the required modifications to what appears in [9] are very simple.
In what follows, any reference to a definition or result in [9] involving a language
actually refers to the appropriate typed version.

To obtain the simplification theorem, we have found it necessary to restrict \( D \) to
be a definite database. A definite database clause is a database clause that has the
form \( A \leftarrow A_1 \land \ldots \land A_n \), where \( A_1, \ldots, A_n \) are atoms. A definite database is a
database that consists of definite database clauses only. The reason for this restric-
tion is that the proof depends crucially on the monotonicity of the mapping \( T_D \)
(defined below) associated with \( D \). Note that, by Propositions 5.1 and 14.3 of [9],
\( \text{comp}(D) \) is consistent if \( D \) is definite.

Suppose \( L \) is the typed language underlying the database \( D \). We make the
assumption throughout that, whatever changes \( D \) may undergo, \( L \) remains fixed.
Thus, for example, adding a new clause to \( D \) does not introduce new constants into
the language. This is effectively the assumption that is made in [13].

Implementing the simplification method involves computing two sets of atoms,
computing two sets of substitutions by unifying atoms in the sets with atoms in an
integrity constraint, and evaluating corresponding instances of the integrity con-
straint. We begin with the definition of the appropriate sets of atoms.

**Definition.** Let \( D \) and \( D' \) be definite databases such that \( D \subseteq D' \). We define the set
\( \text{atom}_{D,D'} \) inductively as follows:

\[
\text{atom}_D^{0} = \{ A : A \leftarrow A_1 \land \ldots \land A_m \in D' \setminus D \},
\]

\[
\text{atom}_{D,D'}^{n+1} = \{ A\theta : A \leftarrow A_1 \land \ldots \land A_m \in D, B \in \text{atom}_{D,D'}^n, \theta \text{ is the mgu of some } A_i \text{ and } B \},
\]

\[
\text{atom}_{D,D'} = \bigcup_{n \geq 0} \text{atom}_{D,D'}^n.
\]

To motivate the above definition, consider the case when we add a fact \( A \) to a
database \( D \) to obtain a database \( D' \). An important task of the simplification method
is to capture the difference between a model for \( \text{comp}(D') \) and a model for
\( \text{comp}(D) \). In the case that \( D \) is a relational database, we see that \( \text{atom}_{D,D'} \) is \( \{ A \} \),
which is precisely the difference between a model for \( \text{comp}(D) \) and a model for
\( \text{comp}(D') \). (In this case the models are essentially unique [15].) For a deductive
database, the presence of rules means that the difference between the models could be larger. However, as we shall see, atom\(_{D,D'}\) can still be used to capture the difference between the two models.

A preinterpretation of a database \(D\) is an interpretation of \(D\) that omits the assignments of relations to predicates [9, p. 71].

**Definition.** Let \(J\) be a preinterpretation of a database \(D\), \(V\) a variable assignment \(\text{wrt} \ J\), and \(A\) an atom. Suppose \(A\) is \(p(t_1,\ldots,t_n)\), and \(d_1,\ldots,d_n\) are the term assignments of \(t_1,\ldots,t_n\) \(\text{wrt} \ J\) and \(V\). We call \(A_{J,V} = p(d_1,\ldots,d_n)\) the \(J\)-instance of \(A\) \(\text{wrt} \ V\). Let \([A]_J = \{A_{J,V} : V\text{ is a variable assignment \(\text{wrt} \ J\)}\}\. We call each element of \([A]_J\) a \(J\)-instance of \(A\). We also call each \(p(d_1,\ldots,d_n)\) a \(J\)-instance. Each interpretation based on \(J\) can now be identified with a subset of \(J\)-instances as in [9, p. 72].

**Definition.** Let \(D\) and \(D'\) be definite databases such that \(D \subseteq D'\) and \(J\) a preinterpretation of \(D\). We define \(\text{inst}\_{D,D',J} = \bigcup_{A \in \text{atom}_{D,D'} [A]_J}\). The essential property of \(\text{inst}\_{D,D',J}\) is presented in Lemma 6 and used in Theorem 3 to capture the difference between models of \(\text{comp}(D)\) and \(\text{comp}(D')\).

**Definition.** Let \(J\) be a preinterpretation of a definite database \(D\). Let \(I\) be an interpretation based on \(J\). Then \(T_D^D(I) = \{A_{J,V} : A \leftarrow A_1 \wedge \cdots \wedge A_n \in D, V\text{ is a variable assignment \(\text{wrt} \ J\)}\}\. It is convenient to suppress the \(J\) and denote this mapping by \(T_D^D\). We also define \(E = \bigcup_{x \in \text{atom}_{D,D'} [A]_J}\). Subsequent use of \(E\) ensures that all models considered are normal.

**Lemma 6.** Let \(D\) and \(D'\) be definite databases such that \(D \subseteq D'\). Let \(J\) be a preinterpretation of \(D\).

(a) Let \(M'\) be an interpretation based on \(J\) for \(D'\) such that \(M' \cup E\) is a model for \(\text{comp}(D')\). Then we have \(M' \setminus T_D^D(M') \subseteq \text{inst}\_{D,D',J}\) for every ordinal \(\alpha\).

(b) Let \(M\) be an interpretation based on \(J\) for \(D\) such that \(M \cup E\) is a model for \(\text{comp}(D)\). Then we have \(T_D^D(M) \setminus M \subseteq \text{inst}\_{D,D',J}\) for every ordinal \(\alpha\).

**Proof.** (a): First note that \(M'\) is a fixpoint of \(T_D^D\), by Proposition 14.3 of [9]. Hence \(T_D^D(M') \subseteq M'\), and so \(T_D^D(M')\) is defined for every ordinal \(\alpha\). The proof is by transfinite induction. We consider the following two cases.

**Case 1:** \(\alpha\) is a limit ordinal. The case \(\alpha = 0\) is trivial. Otherwise, \(M' \setminus T_D^D(M') = M' \setminus \bigcap_{\beta < \alpha} T_D^D(M') = \bigcup_{\beta < \alpha} [M' \setminus T_D^D(M')] \subseteq \text{inst}_{D,D',J}\), by the induction hypothesis.

**Case 2:** \(\alpha\) is a successor ordinal. The case \(\alpha = 1\) is trivial. Otherwise, note that \(M' \setminus T_D^D(M') = [M' \setminus T_D^D(M')] \cup [T_D^D(M') \setminus T_D^D(M')]\). Suppose that \(B \in T_D^D(M') \setminus T_D^D(M')\). Then there exists a clause \(A \leftarrow A_1 \wedge \cdots \wedge A_n\) in \(D\) such
that, for some variable assignment $V$ wrt $J$ and for some $i$, $B_i$ is the $J$-instance of $A$ wrt $V$, and $B_i \in M' \setminus T^{D'}_D(M')$. Thus, by the induction hypothesis, $B_i \in \text{inst}_{D,D'}$. Hence $B_i$ is also a $J$-instance of some $C \in \text{atom}_{D,D'}$. By Lemma 15.1(a) of [9], $A_i$ and $C$ are unifiable with mgu $\theta - \{x_i/r_1, \ldots, x_m/r_m\}$, say. Since $C \in \text{atom}_{D,D'}$ and $A_i \theta = C\theta$, we have that $A\theta \in \text{atom}_{D,D'}$. By Lemma 15.1(b) of [9], the variable assignment that maps $A_i$ and $C$ to $B_i$ also maps $x_i$ and $r_i$ to the same domain element, for each $j$. Hence $B_i$ is also a $J$-instance of $A\theta$, and so $B \in \text{inst}_{D,D',J}$.

\[\text{(b): The proof is similar to part (a). } \Box\]

Let $W$ be a formula in prenex conjunctive normal form. If a negative literal $\sim A$ appears in some conjunct of the matrix of $W$, we say that $A$ is a negated atom in $W$. If a positive literal $A$ appears in some conjunct of the matrix of $W$, we say that $A$ is an atom in $W$.

The addition of a clause $C$ to a database $D$ may cause a $J$-instance of a negated atom in $W$ that is not in a model $M$ of $\text{comp}(D)$ to be in a model $M'$ of $\text{comp}(D \cup \{C\})$, and thus cause $W$ to be false wrt $M'$. The set $\Theta$ in Theorem 3 below describes all the ways in which this may occur, and which instances of $W$ must hence be checked. A similar comment applies to deletions and the set $\Psi$. We now state the simplification theorem for integrity constraints.

**Theorem 3.** Let $D$ and $D'$ be definite databases, and $t$ a transaction whose application to $D$ produces $D'$. Suppose $t$ consists of a sequence of deletions followed by a sequence of additions and that the application of the sequence of deletions to $D$ produces the intermediate database $D''$. Let $W = \forall x_1 \cdots \forall x_n W'$ be an integrity constraint in prenex conjunctive normal form. Suppose $D$ satisfies $W$. Let $\Theta = \{ \theta : \theta$ is the restriction to $x_1, \ldots, x_n$ of an mgu of a negated atom in $W$ and an atom in $\text{atom}_{D',D'} \}$ and $\Psi = \{ \psi : \psi$ is the restriction to $x_1, \ldots, x_n$ of an mgu of an atom in $W$ and an atom in $\text{atom}_{D',D'} \}$. Then we have the following properties:

(a) $D'$ satisfies $W$ iff $D'$ satisfies $\forall(W')$ for all $\phi \in \Theta \cup \Psi$.

(b) If $D' \cup \{ \leftarrow \forall(W') \}$ has an SLDNF-refutation for all $\phi \in \Theta \cup \Psi$, then $D'$ satisfies $W$.

(c) If $D' \cup \{ \leftarrow \forall(W') \}$ has a finitely failed SLDNF-tree for some $\phi \in \Theta \cup \Psi$, then $D'$ violates $W$.

**Proof.** (a): Suppose $D'$ satisfies $\forall(W')$ for all $\phi \in \Theta \cup \Psi$. Let $M'$ be an interpretation for $D'$ based on $J$ such that $M' \cup E$ is a model for $\text{comp}(D')$. Since $T^{D'}(M') \subseteq M'$ and $T^{D'}$ is monotonic, by Propositions 5.3 and 14.3 of [9] there exists an ordinal $\alpha$ such that $M'' \cup E$ is a model for $\text{comp}(D'')$, where $M'' = T^{D'}(M')$. Similarly, there exists an ordinal $\beta$ such that $M \cup E$ is a model for $\text{comp}(D)$, where $M = T^{D}_D(M')$. By supposition, $W$ is true wrt $M \cup E$. Let $V$ be a variable assignment wrt $J$. We have to prove that $W'$ is true wrt $M' \cup E$ and $V$. If $V^*$ is a variable assignment that agrees with $V$ on $x_1, \ldots, x_n$, then we say $V^*$ is compatible with $V$. We consider the following two cases.

**Case 1:** For every negated atom $A$ in $W$ and for every $V^*$ compatible with $V$, the $J$-instance $p(d_1, \ldots, d_n)$ of $A$ wrt $V^*$ is not in $M' \setminus M$, and for every atom $B$ in
A BASIS FOR DEDUCTIVE DATABASE SYSTEMS

W and for every V* compatible with V, the J-instance q(e₁, ..., eₘ) of B wrt V* is not in M \ M'. Let A be a negated atom in W, and suppose that, for some V* compatible with V, the J-instance p(d₁, ..., dₙ) of A wrt V* is not in M. By the condition of case 1, we have that p(d₁, ..., dₙ) ∉ M' \ M. Hence p(d₁, ..., dₙ)∉ M'. Let B be an atom in W, and suppose that, for some V* compatible with V, the J-instance q(e₁, ..., eₘ) of B wrt V* is in M. By the condition of case 1, we have that q(e₁, ..., eₘ)∉ M \ M'. Hence q(e₁, ..., eₘ)∈ M'. It follows from this that W' is true wrt M' ∪ E and V.

Case 2: Either there exists a negated atom A in W and a V* compatible with V such that the J-instance p(d₁, ..., dₙ) of A wrt V* is in M' \ M or there exists an atom B in W and a V* compatible with V such that the J-instance q(e₁, ..., eₘ) of B wrt V* is in M \ M'. Suppose there exists a negated atom A in W and a V* compatible with V such that the J-instance p(d₁, ..., dₙ) of A wrt V* is in M' \ M. Then p(d₁, ..., dₙ)∈ M' \ M' and, by Lemma 6(a), p(d₁, ..., dₙ)∈ inst_{d', D', J}. Thus, p(d₁, ..., dₙ) is also a J-instance of an atom F∈ atom_{D', D, J}. By Lemma 15.1(a) of [9], A and F are unifiable with mgu θ', say. Let θ be the restriction of θ' to x₁, ..., xₙ. By supposition, ∀(Wθ') is true wrt M' ∪ E. It then follows from Lemma 15.1(b) of [9] that W' is true wrt M' ∪ E and V. On the other hand, suppose there exists an atom B in W and a V* compatible with V such that the J-instance q(e₁, ..., eₘ) of B wrt V* is in M \ M'. Then q(e₁, ..., eₘ)∈ M \ M' and, by Lemma 6(b), q(e₁, ..., eₘ)∈ inst_{D', D, J}. Thus, q(e₁, ..., eₘ) is also a J-instance of an atom G∈ atom_{D', D}. By Lemma 15.1(a) of [9], B and G are unifiable with mgu ψ', say. Let ψ be the restriction of ψ' to x₁, ..., xₙ. By supposition, ∀(Wψ') is true wrt M' ∪ E. It then follows from Lemma 15.1(b) of [9] that W' is true wrt M' ∪ E and V.

(b): This part follows immediately from Theorem 1 and part (a).
(c): Suppose D'∪{ ← ∀(Wφ)} has a finitely failed SLDNF-tree, for some φ ∈ Θ ∪ Ψ. By Theorem 1 of [10] and Lemma 1, ¬∀(Wφ) is a logical consequence of comp(D'). Hence W is not a logical consequence of comp(D'), and so D' violates W. □

Theorem 3 has an immediate corollary for the situation when the transaction consists of a single addition.

Corollary 1. Let D be a definite database, C a definite database clause, and D' = D ∪ {C}. Let W = ∀x₁ · · · ∀xₙW' be an integrity constraint in prenex conjunctive normal form. Suppose D satisfies W. Let Θ = {θ : θ is the restriction to x₁, ..., xₙ of an mgu of a negated atom in W and an atom in atom_{D, D'}}. Then we have the following properties:

(a) D' satisfies W iff D' satisfies ∀(Wθ) for all θ ∈ Θ.
(b) If D'∪{ ← ∀(Wθ)} has an SLDNF-refutation for all θ ∈ Θ, then D' satisfies W.
(c) If D'∪{ ← ∀(Wθ)} has a finitely failed SLDNF-tree for some θ ∈ Θ, then D' violates W.
Similarly, Theorem 3 has a corollary for the case when the transaction consists of a single deletion.

**Corollary 2.** Let $D$ be a definite database, $C$ a definite database clause in $D$, and $D' = D \setminus \{C\}$. Let $W \equiv \forall x_1 \cdots \forall x_n W'$ be an integrity constraint in prenex conjunctive normal form. Suppose $D$ satisfies $W$. Let $\Psi = \{ \psi : \psi$ is the restriction to $x_1, \ldots, x_n$ of an mgu of an atom in $W$ and an atom in $\text{atom}_{D', D} \}$. Then we have the following properties:

(a) $D'$ satisfies $W$ iff $D'$ satisfies $\forall(W' \psi)$ for all $\psi \in \Psi$.

(b) If $D' \cup \{ \leftarrow \forall(W' \psi) \}$ has an SLDNF-refutation for all $\psi \in \Psi$, then $D'$ satisfies $W$.

(c) If $D' \cup \{ \leftarrow \forall(W' \psi) \}$ has a finitely failed SLDNF-tree for some $\psi \in \Psi$, then $D'$ violates $W$.

Some discussion of Theorem 3 and its corollaries is appropriate. Theorem 3 is our simplification theorem for checking the integrity constraints when updating a database. It shows that the implementation of the simplification method involves calculating $\text{atom}_{D', D}$ and $\text{atom}_{D', D'}$, computing $\Theta$ and $\Psi$, and then evaluating each query $\leftarrow \forall(W' \phi)$, where $\phi \in \Theta \cup \Psi$. The assumption that $W$ is in prenex conjunctive normal form does not result in any loss of generality, since any formula can be transformed into an equivalent one which is in this form.

Theorem 3 is essentially the generalization to deductive databases of a result for relational databases due to Nicolas (Theorem 3 of [13]). To see that our result is indeed a generalization, note that Reiter [15] has proved a theorem which demonstrates the equivalence of the "model-theoretic" view of relational databases used in [13] and the "proof-theoretic" view used here. In the case of relational databases, $\text{atom}_{D', D}$ is simply the facts being deleted and $\text{atom}_{D', D'}$ is simply the facts being added in the transaction. This is exactly the situation that Nicolas considered.

Some special cases of Theorem 3 are of interest. If $\Theta \cup \Psi$ is empty, then the corresponding integrity constraint $W$ can be eliminated from further consideration, since Theorem 3 shows that $D'$ satisfies $W$. If $\Theta \cup \Psi$ contains the identity substitution, then no simplification of $W$ is possible. Nicolas [13] also studied various refinements of the basic idea which could lead to optimizations of the implementation. We do not discuss these optimizations here except to note that all of them are equally applicable to deductive databases.

The key to an efficient implementation of the simplification theorem is to find an efficient way to calculate $\text{atom}_{D', D}$ for $D \subseteq D'$. We emphasize that this calculation only involves the rules and not the facts in $D$. This is an important point because, even for a large deductive database, the number of rules is likely to be very much smaller than the number of facts. In particular, the rules are likely to be kept in main memory, so that access to the disk during the calculating of this set is obviated.

We now briefly consider one aspect of this computation. In principle, the calculation of $\text{atom}_{D', D}$ involves the calculation of infinitely many sets $\text{atom}_{D', D'}^n$ for $n \geq 0$. However, in practice we can often use a stopping rule to terminate the computation after only finitely many steps. Suppose we have just computed $A \in \text{atom}_{D', D'}^m$ and we note that $A$ is an instance of an atom in some $\text{atom}_{D', D'}^n$, where $0 \leq m \leq n$. Then the above proof shows that we can delete $A$ from $\text{atom}_{D', D'}^m$ and still
obtain Theorem 3. The stopping rule is then as follows. If after deletions in this manner, some atom_{D'}^i becomes empty, then terminate the computation of atom_{D'} and use the set S of atoms computed thus far in place of atom_{D'D'}^i. A further refinement is to delete from S any atom which is an instance of another atom in S. The example below illustrates the application of this stopping rule.

Example. Let D be the database
\begin{align*}
\text{ancestor}(x, y) & \leftarrow \text{parent}(x, z), \text{ancestor}(z, y) \\
\text{ancestor}(x, y) & \leftarrow \text{parent}(x, y) \\
\text{parent}(x, y) & \leftarrow \text{mother}(x, y) \\
\text{parent}(x, y) & \leftarrow \text{father}(x, y)
\end{align*}
together with facts for the predicates mother and father. Let C be the clause
\[
\text{mother}(\text{mary}, \text{bill}) \leftarrow
\]
and let D' = D ∪ {C}. Then we obtain
\begin{align*}
\text{atom}_{D', D'}^0 &= \{ \text{mother}(\text{mary}, \text{bill}) \}, \\
\text{atom}_{D', D'}^1 &= \{ \text{parent}(\text{mary}, \text{bill}) \}, \\
\text{atom}_{D', D'}^2 &= \{ \text{ancestor}(\text{mary}, \text{bill}), \text{ancestor}(\text{mary}, y) \}, \\
\text{atom}_{D', D'}^3 &= \{ \text{ancestor}(x, \text{bill}), \text{ancestor}(x, y) \}, \\
\text{atom}_{D', D'}^4 &= \{ \text{ancestor}(x, \text{bill}), \text{ancestor}(x, y) \}.
\end{align*}
At this point, we can apply the stopping rule. Thus, in place of atom_{D', D'}^i, we can use the set S = \{ \text{mother}(\text{mary}, \text{bill}), \text{parent}(\text{mary}, \text{bill}), \text{ancestor}(x, y) \}.

It should be clear that checking that a database still satisfies its integrity constraints after an update can be very expensive. The reason is that the presence of rules in the database means that the extensions of many predicates, other than the one directly affected by the update, could be changed. In fact, it is easy to construct examples where the addition of a single fact, whose predicate does not appear in any integrity constraint, will require that every integrity constraint be checked without any possible simplification.

Furthermore, in the worst case, atom_{D', D'}^i can contain an infinite set of "independent" atoms. In this case, we have clearly made no simplification at all. However, it would appear that with the kind of database one might find in practice, this is not likely to happen. For example, suppose a definite database D is hierarchical. Then it is clear we are guaranteed that the stopping rule will be applicable after finitely many steps. In fact, as the above example shows, the stopping rule can be applicable even if D contains recursive rules.

5. OPEN PROBLEMS

There is now a substantial theory of deductive database systems, which is summarized in [7] and extended in the current paper. There remain, however, many interesting unsolved problems which deserve investigation. We list below some of the more important of these that are relevant to our approach.
Consistency. What is the largest recursive class of databases $D$ such that comp($D$) is consistent? Known conditions under which comp($D$) is consistent are that $D$ is definite or that $D$ satisfies the condition in Theorem 3 of [16]. Also, if $D$ is hierarchical, it is easy to construct a typed Herbrand model for comp($D$).

Completeness. What are the most general conditions under which all (ground) correct answer substitutions for comp($D$) $\cup \{Q\}$ can be computed by our query evaluation process? If there are only finitely many answers to $Q$, we require the system to compute them all and then halt; otherwise, we require that, for each answer, the system eventually compute that answer and then continue. Several related completeness results are given in [1], [9, §8, §16], and [16].

Simplification of integrity constraints for arbitrary databases. How can Theorem 3 be extended to arbitrary (nondefinite) databases? The difficulty of this problem arises from the fact that $T_D$ is not monotonic in this case. Alternatively, what is the largest recursive class of databases for which the appropriate version of Theorem 3 holds?

Finiteness of atom$_{D,D'}$. What are the most general conditions on $D$ and $D'$ ($D \subseteq D'$) which ensure that atom$_{D,D'}$ is finite?

Implementation of integrity constraint checking. How can integrity constraint checking based on Theorem 3 be implemented efficiently? In particular, how can atom$_{D,D'}$ be evaluated efficiently for arbitrary $D$ and $D'$ ($D \subseteq D'$)? The stopping rule given above is clearly relevant, but additional methods are also required.

Other open problems related to deductive database systems are discussed in [7].

We thank Liz Sonenberg for her helpful comments on a draft of this paper.

REFERENCES

17. Topor, R. W., Keddis, T., and Wright, D. W., Deductive Database Tools, TR 84/7, Dept. of Computer Science, Univ. of Melbourne, 1984.