ON DERIVED DEPENDENCIES AND CONNECTED DATABASES

PHILIP W. DART

▷ This paper introduces a new class of deductive databases (*connected databases*) for which SLDNF-resolution never flounders and always computes ground answers. The class of connected databases properly includes that of allowed databases. Moreover the definition of connected databases enables evaluable predicates to be included in a uniform way. An algorithm is described which, for each predicate defined in a normal database, derives a propositional formula (*groundness formula*) describing dependencies between the arguments of that predicate. Groundness formulae are used to determine whether a database is connected. They are also used to identify goals for which SLDNF-resolution will never flounder and will always compute ground answers on a connected database. ◁

1. INTRODUCTION

In the context of relational databases, it has long been realized that only some first order formulae are reasonable queries; these formulae have desirable properties such as domain independence and finiteness. Because these properties are undecidable, various decidable subclasses, of at least the domain independent formulae, have been proposed. These classes include range separable [4], range restricted [17, 8], evaluable [9], allowed [21], and safe [24].[1] A survey of the decidability of domain independence and finiteness for different query languages is given in [11]. In the context of deductive databases, additional properties of databases and queries are desirable, such as nonfloundering and, in some cases, ground answers. Proposed classes of databases with these undecidable properties include allowed databases [3, 21, 22]. In order to ensure that certain properties hold, most of these

Address correspondence to Philip W. Dart, Information Technology Division, Defence Science and Technology Organisation, P.O. Box 1600, Salisbury 5108, Australia.
Received June 1988; accepted June 1989.
[1]Safety as defined in [23] is a semantic concept and not obviously decidable.

classes forbid or restrict the use of one or more of negation, functions, recursive predicates, and evaluable predicates. These restrictions are severe, since they exclude predicates commonly used in logic programming for such fundamental tasks as list manipulation and arithmetic.

Some properties of the predicates of a deductive database relate to the propagation of bindings between the arguments of predicates during resolution. The manner in which bindings are propagated may be described as dependencies. Propositional formulae called *groundness formulae* can be derived for the predicates of a deductive database to describe these dependencies. An algorithm that derives groundness formulae for a deductive database with evaluable predicates is described; testing these groundness formulae provides a sufficient condition for determining whether the database is *connected*. The class of connected databases properly includes the class of allowed databases, and connected databases have some of the desirable properties of allowed databases. Groundness formulae can be used to identify goals for which SLDNF-resolution never flounders and computes ground answers on a connected database.

Section 2 of this paper presents some of the background and motivation for this type of analysis, including discussion of the definition of the class of allowed databases and the application of modes [26] in defining an extension to this class. Section 3 defines a class of propositional formulae that can be used to describe dependencies. Section 4 defines groundness formulae, relates them to the clausal definitions of predicates in a normal database, and gives an algorithm for deriving groundness formulae for a normal database. In Section 5, the class of connected deductive databases is defined, leading to the proof of a theorem that identifies goals for which SLDNF-resolution will generate ground answers and not flounder on this class of databases. This section also contains a proof of the theorem that the class of connected databases properly includes the class of allowed databases.

Unless otherwise stated, the notation and terminology used in this paper is consistent with [14].

2. ALLOWED DATABASES AND OTHER PRELIMINARIES

This section considers allowed formulae and databases and some of their properties. Throughout this paper, it is assumed that every database and goal is expressed in some first order language, with equality, containing only finitely many constants and function symbols.

Definition 2.1. A *normal goal* has the form $\leftarrow L_1 \wedge \cdots \wedge L_n$ where $L_1, \ldots, L_n$ are literals. A *database clause* has the form $A \leftarrow L_1 \wedge \cdots \wedge L_n$ where A is an atom and $L_1, \ldots, L_n$ are literals. A *normal database* is a finite set of database clauses.

Definition 2.2. A predicate p in a database D is *extensional* if all clauses in D defining p are ground unit clauses. A predicates p in a database D is *intensional* if some clause in D defining p is not a ground unit clause. A predicate p in a database D is *evaluable* if p is not defined by a set of clauses in D; occurrences of atoms of p encountered during query evaluation will be evaluated directly.

Examples of evaluable predicates include arithmetic and relational predicates such as *plus*, >=, and ~=. Evaluable predicates are often viewed as being defined by (possibly infinite) sets of ground unit clauses.

Allowed formulae were introduced in [3]. The following definition of allowed databases, applicable to normal databases and goals, is adapted from that given in [13]. This definition permits allowed databases to contain restricted occurrences of the equality predicate.

Definition 2.3. A variable x is *pos* (positive) in a formula if one of the following cases holds:

x is pos in $p(t_1,\ldots,t_n)$ if x occurs in $p(t_1,\ldots,t_n)$ and p is not $=$.

x is pos in $x=t$ or $t=x$ if t is a ground term.

x is pos in $L_1 \wedge \cdots \wedge L_n$ if x is pos in some literal L_i, $1 \leq i \leq n$.

Definition 2.4. A normal goal G is *allowed* if every variable that occurs in G is pos in the body of G. A database clause $A \leftarrow L_1 \wedge \cdots \wedge L_n$ is *admissible* if every variable that occurs in the clause either occurs in the head A or is pos in the body $L_1 \wedge \cdots \wedge L_n$. A database clause $A \leftarrow L_1 \wedge \cdots \wedge L_n$ is *allowed* if every variable that occurs in the clause is pos in the body $L_1 \wedge \cdots \wedge L_n$. If D is a normal database and G is a normal goal, than $D \cup \{G\}$ is *allowed* if the following conditions are satisfied:

(a) Every clause in D is admissible.

(b) Every clause in the definition of a predicate occurring in a positive literal in the body of G or the body of a clause in D is allowed.

(c) G is allowed.

Note that every allowed unit clause is ground and that every allowed clause is admissible.

Three definitions are given next, followed by a theorem that expresses the desirable operational properties that are an important reason for restricting attention to the class of allowed databases [13].

Definition 2.5. A literal is *sufficiently instantiated* if it is ground or positive.

Definition 2.6. A *safe computation rule* is a function mapping normals goals (containing at least one sufficiently instantiated literal) to literals such that the value of the function for such a goal is always a sufficiently instantiated literal (called the *selected* literal) in that goal.

Definition 2.7. Let D be a normal database, G a normal goal, and R a safe computation rule. The evaluation of $D \cup \{G\}$ via *R flounders* if at some point in the evaluation a goal is reached which contains no sufficiently instantiated literals.

Theorem. Let D be a normal database, G a normal goal, and R a safe computation rule. If $D \cup \{G\}$ is allowed, then the following properties hold:

(a) *Query evaluation of $D \cup \{G\}$ via R does not flounder.*
(b) *Every R-computed answer for $D \cup \{G\}$ is a ground substitution for all free variables in G.*

Unfortunately, the class of allowed formulae is severely restricted in several areas. The definition permits only a restricted form of equality in which one argument must be ground and therefore excludes some reasonable formulae such as the following example:

$$p(x) \quad \wedge \quad x=y \quad \wedge \quad \sim q(y).$$

Also, the clauses in the common definitions of predicates such as *append* and *member* are not allowed. Despite the fact such clauses are admissible, the recursive calls in the clauses exclude them from being included in an allowed database. Furthermore, the definition does not consider evaluable predicates such as the arithmetic predicate *plus*. While many evaluable predicates could be redefined operationally to permit their inclusion in the class of allowed formulae, the resulting computational behavior would be undesirable in many cases. For example, the predicate *plus* could be defined by an infinite set of ground unit clauses (ignoring finiteness constraints on the database), but this is not a practical solution.

The simple and concise definition of the class of allowed databases, given above, is possible because of the restrictive nature of the class. Although this definition could be extended to permit other uses of equality, it is preferable to define an extended class that permits equality and evaluable predicates in an uniform manner. It is shown below that dependency information relating the arguments of predicates can be used to define an extension of this class that permits evaluable predicates and less restrictive use of equality while preserving the desirable properties of the class.

Modes

It has long been recognized, in logic programming, that many evaluable and intensional predicates are not used to instantiate all of their arguments. For this reason, the notion of *modes* was introduced to describe the ways in which a predicate will be used in a PROLOG program [26]. Modes are states of instantiation of the arguments of a predicate; an N-ary predicate can have a set of N-ary tuples of modes associated with it describing the states of instantiation in which the predicate will be called. The automatic generation of modes for PROLOG predicates [15] and logic programs [6, 7, 27] has been discussed elsewhere.

As discussed in the previous section, query evaluation for allowed databases has desirable properties that result from the propagation of bindings under SLDNF-resolution. To permit evaluable predicates in an extension of the class of allowed databases, the propagation of bindings must be considered for predicates defined by clauses containing evaluable atoms. To extend allowed databases, the state of instantiation of a literal after it has been evaluated is of interest; modes can be used to represent such information. A mode indicating definite instantiation to a ground term is of obvious importance where SLDNF-resolution using a safe computation rule is to be applied. In this paper, the exclusive and exhaustive set of the following two modes will be considered: g (always instantiated to a ground term) and $\bar{g}$ (not always instantiated to a ground term). Such modes can be considered as propositions about the state of instantiation of an argument of a predicate. This set of modes can be used to provide a simple but approximate groundness analysis of deductive databases.[2]

These propositional modes can be used in n-ary tuples to describe possible states of instantiation of the arguments of an n-ary atom after evaluation during SLDNF-resolution. For most predicates, there are states from which an atom of that predicate will generate no further bindings, even in a successful or floundering SLDNF-computation; these states represent information that is useful in the analysis of a deductive database. For example, the mode tuples for the standard ternary predicate *append* would be $\langle \bar{g}, \bar{g}, \bar{g} \rangle$, $\langle g, \bar{g}, \bar{g} \rangle$, $\langle \bar{g}, g, \bar{g} \rangle$ and $\langle g, g, g \rangle$. The reason that, for example, the mode tuple $\langle \bar{g}, \bar{g}, g \rangle$ has not been included is that a call to *append* with the last argument ground will always ground its other two arguments in any successful or floundering SLDNF-computation. Obviously, the mode tuple $\langle g, g, \ldots, g \rangle$ would be included for any predicate and would be the only mode tuple applicable to predicates defined only by allowed clauses.

Using g_i to denote the proposition "the ith argument of *append* is instantiated to a ground term", the set of the mode tuples given above for *append* can also be expressed as the propositional formula

$$(\bar{g}_1 \wedge \bar{g}_2 \wedge \bar{g}_3) \vee (g_1 \wedge \bar{g}_2 \wedge \bar{g}_3) \vee (\bar{g}_1 \wedge g_2 \wedge \bar{g}_3) \vee (g_2 \wedge g_2 \wedge g_3)$$

in disjunctive normal form. Transforming this into conjunctive normal form gives a set of definite clauses. Rewriting each clause as an implication gives

$$(g_3 \rightarrow g_1) \wedge (g_3 \rightarrow g_2) \wedge (g_1 \wedge g_2 \rightarrow g_3),$$

which can be interpreted as describing how *append* propagates bindings. This formula can be further abbreviated to $g_1 \wedge g_2 \leftrightarrow g_3$ and can be read informally as "if a call to *append* with its first two arguments ground succeeds, it will return its third argument ground, and vice versa". A propositional formula that describes the dependencies between the arguments of a predicate in this way is a *groundness formula*. The conjuncts of a groundness formula are similar to *safety dependencies* [28]. Groundness formulae are used in Section 4 as a basis for extending the class of allowed formulae.

3. DEPENDENCY FORMULAE

This section describes *dependency formulae*, a class of propositional formulae that can be used to describe dependencies, and operations on dependency formulae. *Groundness formulae*, to be introduced in Section 4, are dependency formulae with the property that they can be used to describe the propagation of bindings during

[2]By using information about the possible (partially ground) structures to which a variable may be instantiated, a more precise analysis can be performed. One approach is to use typing (for example polymorphism [5]) augumented by modes. This approach is used in [18, 1].

the evaluation of a goal on a database. Dependency formulae can be used to describe other dependencies such as type dependencies [5].

In the context of pure logic programming, it is not possible for a variable to be instantiated as a consequence of some other variable being uninstantiated.[3] Thus a condition C_i, for the proposition g_i ("the term t_i is ground") to be true, can be expressed as a propositional formula based only on whether certain other terms are ground. Therefore, C_i can be expressed as a *positive* propositional formula, that is, a formula in which the only operators are conjunction and disjunction. This motivates the following definition.

Definition 3.1. Let P be the finite set of propositions $\{g_1, \dots, g_n\}$, and $C_1, \dots, C_n$ be positive propositional formulae on $P \setminus \{g_i\}$. Then a *dependency formulae* on P is a propositional formula on P of the form

$$\bigwedge_{i=1}^{n} (C_i \to g_i).$$

Example 3.1. The following are examples of dependency formulae. Each dependency formula is on the set of propositions appearing in the formula:

$$\mathit{false} \to g_1, \qquad \mathit{true} \to g_1, \qquad (g_2 \to g_1) \wedge (g_1 \to g_2),$$
$$(\mathit{false} \to g_1) \wedge (\mathit{false} \to g_2) \wedge (g_1 \vee g_2 \to g_3).$$

These formulae can be simplified, respectively, to

$$\mathit{true}, \qquad g_1, \qquad g_1 \leftrightarrow g_2 \qquad g_1 \vee g_2 \to g_3.$$

The following, as will be demonstrated by Proposition 3.1, are not dependency formulae:

$$\mathit{false}, \qquad \bar{g}_1, \qquad g_1 \vee g_2, \qquad g_1 \wedge g_2 \to g_3 \vee g_4.$$

Note that the condition *true* denotes an empty conjunction and the condition *false* denotes an empty disjunction. Throughout this paper, dependency formulae are expressed in a simplified form where convenient, as illustrated in Example 3.1.

It will be useful to consider a model-theoretic characterization of dependency formulae. A model for a propositional formula F on a set of propositions P is identified with a subset of P. An equivalence class of propositional formulae can be characterized by the set of models for which the members of that class are true. The set of models that characterizes an equivalence class of dependency formulae is described by the following proposition.

Proposition 3.1. Let F be a propositional formula on P. Then F is logically equivalent to a dependency formula on P if and only if the set M of models for F satisfies:

(a) $P \in M$,

(b) *for all* $I_1 \in M$ *and* $I_2 \in M$, $I_1 \cap I_2 \in M$.

[3]This is a contrast to PROLOG, in which nonlogical predicates such as "$p(X,Y)$:- $var(X)$, $Y = 1$." can be defined.

PROOF. If F is logically equivalent to a dependency formula $\bigwedge_{g_i \in P} C_i \to g_i$, then the set M of models for F satisfies (a) and (b).

(a): Trivial.

(b): Consider two models $I_1 \in M$ and $I_2 \in M$. For any $g_i \in P$, one of the following two cases holds:

(i) $I_1 \models g_i$ and $I_2 \models g_i$. Then $I_1 \cap I_2 \models g_i$ and $I_1 \cap I_2 \models C_i \to g_i$.

(ii) $I_1 \not\models g_i$ or $I_2 \not\models g_i$. Assume $I_1 \not\models g_i$. Since $I_1 \models C_i \to g_i$ then $I_1 \not\models C_i$. C_i is positive and therefore monotonic; thus $I_1 \cap I_2 \not\models C_i$ and $I_1 \cap I_2 \models C_i \to g_i$. (Similarly if $I_2 \not\models g_i$.)

Therefore $I_1 \cap I_2 \models F$ and (b) holds.

If the set M of models for F satisfies (a) and (b), then F is logically equivalent to a dependency formula on P. The following proof is adapted from the proof given for [2, Theorem 1.2.18]. Let ψ be the conjunction of all propositional formulae of the form

$$\bigvee_{g_i \in P_j} \sim g_i \vee g_j,$$

where $P_j \subseteq P \setminus \{g_j\}$, that hold for every model in M. ψ can be rewritten trivially as a dependency formula F. By definition, for all $I \in M$, $I \models \psi$. It is sufficient to show that for all $I \subseteq P$, if $I \models \psi$ then $I \in M$.

Consider $I \subseteq P$ such that $I \models \psi$. If $I = P$ then $I \in M$ by (a). If $I \subset P$, then it is shown that I can be expressed as the intersection of members of M.

For each $g_j \notin I$, let ξ_j be the formula $\bigwedge_I g_i \wedge \sim g_j$. Then $I \models \xi_j$, so $I \not\models \sim \xi_j$, where $\sim \xi_j$ is equivalent to $\bigvee_I \sim g_i \vee g_j$. Since $I \models \psi$ and $I \not\models \sim \xi_j$, ξ_j cannot be a conjunct of ψ. Therefore, from the construction of ψ, there exists a model $I_j \in M$ such that $I_j \not\models \xi_j$. It is now shown that I can be constructed as follows:

$$I = \bigcap_{g_j \in P \setminus I} I_j.$$

Consider $g_i \in P$. If $g_i \in I$, then for all j such that $g_j \in P \setminus I$ we have $g_i \in I_j$. If $g_i \notin I$, then $\xi_i \models \sim g_i$ and $g_i \notin I_i$. Therefore I can be constructed as above. Since each $I_j \in M$, we have $I \in M$ by (b), and the result holds. □

A partial order can be defined on the set of dependency formulae over a set of propositions as follows.

Definition 3.2. Let F and F' be dependency formulae over the set of propositions $\{g_1, \ldots, g_n\}$. Then $F' \leq F$ if $F \to F'$ is a tautology.

It follows that F is *maximal* in this ordering if F is logically equivalent to $g_1 \wedge \cdots \wedge g_n$, and F is *minimal* in this ordering if F is logically equivalent to *true*. A maximal dependency formula corresponds to a formula in which each C_i is true, and a minimal dependency formula to one in which each C_i is false.

Definition 3.3. The expression $C[g/C']$ denotes the result of substituting the formula C' for the proposition g throughout the formula C.

Proposition 3.2. *$C_i[g_i/false] \to g_i$ is equivalent to $C_i \to g_i$.*

PROOF. Trivial. □

By Proposition 3.2, the restriction in Definition 3.1 that C_i does not include g_i may be applied without loss of generality. The restriction prevents trivial dependencies such as *false* $\to g_1$ from being expressed as, for example, $g_1 \to g_1$.

Definitions 3.4, 3.5, and 3.6 introduce dependency formula operators. These operators, together with the propositions proved below, are used in Section 4 to derive an algorithm for generating groundness formulae for a database. Since the definition of dependency formulae is based on a restriction to the set of propositional formulae, the result of operations on dependency formulae should obviously be propositional formulae for which the restriction holds. This property holds for the operators defined in Definitions 3.4, 3.5, and 3.6, which are statements of conjunction, disjunction, and *exclusion* respectively.

Definition 3.4. Let F be the dependency formula $\bigwedge_{\{i:g_i \in P\}}(C_i \to g_i)$ on P, and F' be the dependency formula $\bigwedge_{\{i:g_i \in P'\}}(C_i' \to g_i)$ on P'. Then $F \dot{\wedge} F'$ is the following dependency formula on $P \cup P'$:

$$\left(\bigwedge_{\{i:g_i \in P \cap P'\}} (C_i \vee C_i' \to g_i)\right) \wedge \left(\bigwedge_{\{i:g_i \in P \setminus P'\}} (C_i \to g_i)\right)$$
$$\wedge \left(\bigwedge_{\{i:g_i \in P' \setminus P\}} (C_i' \to g_i)\right).$$

Example 3.2. $((g_1 \to g_2) \wedge g_4) \dot{\wedge} ((g_3 \to g_2) \wedge (g_3 \to g_4))$ is equivalent to $((g_1 \vee g_3 \to g_2) \wedge g_4)$.

This conjunction operator will be used in Section 4 to derive a groundness formula for a conjunction of literals from groundness formulae for each of the literals.

Proposition 3.3. *$(F \wedge F') \leftrightarrow (F \dot{\wedge} F')$ is a tautology.*

PROOF. Since $C_i \vee C_i' \to g_i$ is equivalent to $(C_i \to g_i) \wedge (C_i' \to g_i)$, the result follows directly from Definition 3.4. □

Although propositional conjunction ($\wedge$) and dependency conjunction ($\dot{\wedge}$) are logically equivalent, dependency conjunction is provided as a syntactic characterization of the conjunction of two dependency formulae.

Definition 3.5. Let F be a dependency formula on P, and F' a dependency formula on P'. Then $F \dot{\vee} F'$ is any dependency formula on $P \cup P'$ with the set of models $\{I | I \models F\} \cup \{I' | I' \models F'\} \cup \{I \cap I' | I \models F \text{ and } I' \models F'\}$.

By Proposition 3.1, a dependency formula $F \dot{\vee} F'$ exists and can be determined.

Example 3.3. $((g_1 \to g_2) \wedge (g_1 \to g_3)) \dot{\vee} ((g_2 \to g_3) \wedge (g_2 \to g_4))$ is equivalent to $(g_1 \wedge g_2 \to g_3)$.

This disjunction operator will be used in Section 4 to derive a groundness formula for a predicate from groundness formulae derived for the body of each of its clauses.

Proposition 3.4. $(F \vee F') \to (F \dot{\vee} F')$ *is a tautology.*

PROOF. Consider an interpretation I for $F \vee F'$. Then $I \models F$ or $I \models F'$. By Definition 3.5, $I \models F \dot{\vee} F'$ and the result follows. □

Example 3.3 shows that the converse of Proposition 3.4 is false.

Proposition 3.5. If $F \to F'$ *is a tautology, then* $(F \dot{\vee} F'') \to (F' \dot{\vee} F'')$ *is a tautology.*

PROOF. Consider an interpretation I for $F \dot{\vee} F''$. Then one of three cases holds:

(i) $I \models F''$. Then $I \models (F' \dot{\vee} F'')$.

(ii) $I \models F$. Since $F \to F'$ is a tautology, $I \models F'$ and $I \models F' \dot{\vee} F''$.

(iii) $I = I_1 \cap I_2$, where $I_1 \models F$ and $I_2 \models F''$. Since $F \to F'$ is a tautology, $I_1 \models F'$. Thus $I_1 \cap I_2 \models F' \dot{\vee} F''$.

The result then follows. □

Definition 3.6. Let F be the dependency formula $\bigwedge_{\{i : g_i \in P\}}(C_i \to g_i)$ on P. Then F *excluding* g_k, $F \backslash g_k$, is the following dependency formula on $P \backslash \{g_k\}$:

$$\bigwedge_{\{i : g_i \in P \backslash \{g_k\}\}} (C_i[g_k/C_k][g_i/\mathit{false}] \to g_i).$$

Example 3.4.

$$\begin{aligned}((g_1 \to g_2) \wedge (g_2 \to g_3)) \backslash g_2 &= (g_2[g_2/g_1][g_3/\mathit{false}] \to g_3)\\ &= (g_1[g_3/\mathit{false}] \to g_3)\\ &= (g_1 \to g_3).\end{aligned}$$

Also

$$\begin{aligned}&((g_2 \wedge g_3 \to g_1) \wedge (g_1 \vee g_4 \to g_2) \wedge (g_4 \to g_3)) \backslash g_1\\ &\quad = ((g_1 \vee g_4)[g_1/g_2 \wedge g_3][g_2/\mathit{false}] \to g_2)\\ &\qquad \wedge (g_4[g_1/g_2 \wedge g_3][g_3/\mathit{false}] \to g_3)\\ &\quad = (((g_2 \wedge g_3) \vee g_4)[g_2/\mathit{false}] \to g_2) \wedge (g_4[g_3/\mathit{false}] \to g_3)\\ &\quad = (g_4 \to g_2) \wedge (g_4 \to g_3).\end{aligned}$$

From Definition 3.6 and Example 3.4, it should be evident that the exclusion operation takes into account the transitivity of dependencies. The exclusion operator will be used in Section 4, in the derivation of a groundness formula for a predicate from groundness formulae for the bodies of its clauses, by excluding dependency information relating to variables that occur in the body of a clause but not the head.

Some properties of the exclusion operator are now proven.

Proposition 3.6. Let F be a dependency formula and I an interpretation for F. Then $I \models F\backslash g_k$ if and only if $I \models F$ or $I \cup \{g_k\} \models F$.

PROOF. Proposition 3.2 is used throughout this proof.

(a) Show that if $I \models F\backslash g_k$ then $I \models F$ or $I \cup \{g_k\} \models F$.

Suppose $I \models C_k$. Then it is shown that $I \cup \{g_k\} \models F$. If $g_i \in I$ then $I \cup \{g_k\} \models C_i \rightarrow g_i$. If $g_i \notin I$, then it is necessary to show that $I \cup \{g_k\} \models \sim C_i$. If g_k does not occur in C_i, the result follows. Now $I \models \sim C_i[g_k/C_k][g_i/\mathit{false}]$. If g_k occurs in C_i, then $I \cup \{g_k\} \models \sim C_i[g_k/C_k][g_i/\mathit{false}]$. But g_k and C_k are true and g_i is false in $I \cup \{g_k\}$. So $C_i[g_k/C_k][g_i/\mathit{false}]$ is just C_i, and $I \cup \{g_k\} \models \sim C_i$.

Suppose $I \models \sim C_k$. Then it is shown that $I \models F$. If $g_i \in I$ then $I \models C_i \rightarrow g_i$. If $g_i \notin I$, then it is necessary to show that $I \models \sim C_i$. If g_k does not occur in C_i, the result follows. Now $I \models \sim C_i[g_k/C_k][g_i/\mathit{false}]$. If g_k occurs in C_i, then $I \models \sim C_i[g_k/C_k][g_i/\mathit{false}]$. But g_i, g_k, and C_k are false in I. So $C_i[g_k/C_k][g_i/\mathit{false}]$ is just C_i and $I \models \sim C_i$.

(b) Show that if $I \models F$ or $I \cup \{g_k\} \models F$ then $I \models F\backslash g_k$.

Suppose $I \models F$. Then it is shown that $I \models C_i[g_k/C_k][g_i/\mathit{false}] \rightarrow g_i$. If $g_i \in I$, then this always holds. If $g_i \notin I$, then if g_k and C_k are both either true or false in I, then this will hold, since $I \models C_i \rightarrow g_i$. Since $I \models C_k \rightarrow g_k$, only the case where $g_i \notin I$, $g_k \in I$, $I \models \sim C_k$, and g_k occurs in C_i remains to be considered. In this case, since g_k (which is true in I) is being replaced by C_k (which is false in I) in C_i (which is positive), it follows that $C_i[g_k/C_k][g_i/\mathit{false}] \rightarrow C_i$, and the result follows.

Suppose $I \cup \{g_k\} \models F$. Then the proof is similar to that when $I \models F$, except that, since g_k does not occur in $F\backslash g_k$, $I \cup \{g_k\} \models F\backslash g_k$ if and only if $I \models F\backslash g_k$. □

Proposition 3.7. Let F and F' be dependency formulae on P. Then given $g_j \in P$ and $g_k \in P$,

(a) *F implies $F\backslash g_k$.*

(b) *$F\backslash g_k\backslash g_j$ is logically equivalent to $F\backslash g_j\backslash g_k$.*

(c) *If F implies F', then $F\backslash g_k$ implies $F'\backslash g_k$.*

PROOF. (a): This result follows directly from Proposition 3.6.

(b): Let I be an interpretation for F. Then, applying Proposition 3.6 twice,

$$\begin{aligned} I \models F\backslash g_k\backslash g_j \quad &\text{iff} \quad (I \models F\backslash g_k) \text{ or } (I \cup \{g_j\} \models F\backslash g_k) \\ &\text{iff} \quad (I \models F) \text{ or } (I \cup \{g_k\} \models F) \text{ or } (I \cup \{g_j\} \models F) \\ &\text{or } (I \cup \{g_j\} \cup \{g_k\} \models F). \end{aligned}$$

Since the right hand side is symmetric with respect to g_k and g_j, the result follows.

(c): Let I be an interpretation for F. Then by Proposition 3.6,

$$\begin{aligned} I \models F\backslash g_k \quad &\text{implies} \quad (I \models F) \text{ or } (I \cup \{g_k\} \models F) && \text{(by Proposition 3.6)} \\ &\text{implies} \quad (I \models F') \text{ or } (I \cup \{g_k\} \models F') && \text{(since } F \text{ implies } F'\text{)} \\ &\text{implies} \quad I \models F'\backslash g_k && \text{(by Proposition 3.6)}, \end{aligned}$$

and the result follows. □

Following Proposition 3.7(b), given $P = \{g_j, \ldots, g_k\}$, $F\backslash P$ may be used to denote $F\backslash g_j \cdots \backslash g_k$.

The next two results reveal how dependency formulae propagate information for conjunction and disjunction respectively. The first result shows that if a new proposition can be proved from a set of propositions and a conjunction of dependency formulae, then there is a "convenient" sequence in which the proof can be performed.

Proposition 3.8. Let $F_1, \ldots, F_l$ be dependency formulae on $\{g_1, \ldots, g_n\}$, and S_1 be a subset of $\{1, \ldots, n\}$. If $\bigwedge_{S_1} g_i \wedge (F_1 \dot{\wedge} \cdots \dot{\wedge} F_l) \rightarrow g_j$ is a tautology, then there exists a sequence of sets $S_1, \ldots, S_{s+1}$, $s \geq 0$, where $j \in S_{s+1}$, with the following property. For all t, $1 \leq t \leq s$, there exists some g_k, $1 \leq k \leq n$, $k \notin S_t$, and some F_m, $1 \leq m \leq l$, such that $S_{t+1} = S_t \cup \{k\}$ and $\bigwedge_{S_t} g_i \wedge F_m \rightarrow g_k$ is a tautology.

PROOF. Let **F** be the formula $\bigwedge_{S_1} g_i \wedge (F_1 \dot{\wedge} \cdots \dot{\wedge} F_l)$. A sequence of sets can be constructed as follows. Set $S = S_1$. While any F_m occurs in **F** such that $\bigwedge_S g_i \wedge F_m \rightarrow g_k$ is a tautology and $k \notin S$, then add k to S. When no such F_m can be chosen, then S is a model for **F**. If $\mathbf{F} \rightarrow g_j$ is a tautology, then g_j must hold for every model for **F**. Therefore $j \in S$, and the sequence required is a subsequence of the sequence constructed above. □

The next result shows that if a new proposition can be proved from a set of propositions and a disjunction of dependency formulae, then any one of the dependency formulae could be used to perform the proof.

Proposition 3.9. Let $F_1, \ldots, F_l$ be dependency formulae on $\{g_1, \ldots, g_n\}$, and S be a subset of $\{1, \ldots, n\}$. If $\bigwedge_S g_i (F_1 \dot{\vee} \cdots \dot{\vee} F_l) \rightarrow g_j$ is a tautology, then for all m, $1 \leq m \leq l$, $\bigwedge_S g_i \wedge F_m \rightarrow g_j$ is a tautology.

PROOF. If $\bigwedge_S g_i \wedge (F_1 \dot{\vee} \cdots \dot{\vee} F_l) \rightarrow g_j$ is a tautology, then $\bigwedge_S g_i \wedge (F_1 \vee \cdots \vee F_l) \rightarrow g_j$ is a tautology by Proposition 3.4. This is equivalent to $(\bigwedge_S g_i \wedge F_1 \rightarrow g_j) \wedge \cdots \wedge (\bigwedge_S g_i \wedge F_l \rightarrow g_j)$ being a tautology. The result then follows. □

4. GROUNDNESS FORMULAE

This section gives a normal definition of groundness formulae followed by definitions and results that relate groundness formulae for a predicate to the groundness formulae for the literals that occur in the bodies of its clauses. The results of this section apply to normal databases and goals with functions. Results for hierarchical databases are presented first, then extended to cover recursion; these results combine into an algorithm for deriving groundness formulae for each predicate defined in a deductive database.

In Section 5, the groundness formulae derived by this algorithm are used to test if a database is *connected*. Connected databases are shown to have some of the desirable properties of allowed databases.

The following definition describes the properties a dependency formula must have to be a groundness formula for a goal. This definition is used to verify that dependency formulae derived by the algorithm are groundness formulae.

Definition 4.1. Let D be a database, G be a goal with variables $x_1, \ldots, x_n$, and $S^\eta = \{i : x_i\eta \text{ is ground}\}$, where η is a substitution for $x_1, \ldots, x_n$. Then a dependency formula F on $\{g_1, \ldots, g_n\}$ is a *groundness formula* for G with respect to D if the following holds: For all substitutions σ and for all j, $1 \leq j \leq n$, if

$$\bigwedge_{S^\sigma} g_i \wedge F \to g_j$$

is a tautology, then for all SLDNF-derivations for $D \cup \{G\sigma\}$ that succeed or flounder with mgu's $\theta_1, \ldots, \theta_d$,

$$x_j\sigma\theta_1 \cdots \theta_d$$

is ground.

$gf_D(G)$ denotes the set of all groundness formulae for G with respect to D.

That is, a dependency formula is a groundness formula for a goal G if it can be used, in conjunction with the set of variables in G ground by some substitution σ, to determine variables in $G\sigma$ that will be ground in all successful or floundering computations for $G\sigma$.

From Definition 4.1, it follows that if F is in $gf_D(G)$, F' is a dependency formula, and $F \to F'$ is a tautology, then F' is in $gf_D(G)$. Also, $\bigwedge_{S^\sigma} g_i \wedge \mathit{true} \to g_j$ is a tautology iff $j \in S^\sigma$ (and therefore $x_j\sigma$ is ground), so *true*, and any minimal dependency formula for G, is in $gf_D(G)$ for any G and D.

This paper is not concerned with deciding whether a given dependency formula is a groundness formula for a given goal and database, but rather with presenting a method for constructing a groundness formula for a given goal and database. From the statement above, members of the *weakest* set of dependency formulae—that is, minimal dependency formulae—give no useful information as to the propagation of bindings. Intuitively, it is desirable to construct a member of the *strongest* possible set of groundness formulae, which provide the most information about which variables will be ground by SLDNF-resolution. The construction described below makes use of dependency information known about predicates to construct groundness formulae that provide this information when possible. For example, if a dependency is known to hold for all clauses in the definition of a predicate, then a groundness formula containing that dependency will be constructed.

A definition that associates groundness formulae with predicates is required.

Definition 4.2. Let D be a database, p an n-ary predicate in D, and F a dependency formula. Then F is a groundness formula for p with respect to D if

F is in $gf_D(\leftarrow p(x_1, \ldots, x_n))$. The symbol $gf_D(p)$ denotes the set of all groundness formulae for p with respect to D.

Thus, $gf_D(p)$ and $gf_D(\leftarrow p(x_1, \ldots, x_n))$ are equivalent. If F is in $gf_D(p)$, then F indicates conditions under which each argument of p is instantiated to a ground term by p, depending on which of the other arguments are ground terms.

Example 4.1. The dependency formula $g_1 \wedge g_2 \leftrightarrow g_3$ is in $gf_D(append)$ for any database D containing the usual definition of the ternary predicate *append*.

The dependency formula given as a groundness formula for *append* in Example 4.1 describes intuitively how *append* behaves with respect to groundness. The statement made in this example will be proven later in this paper.

A dependency formula F will be referred to as being maximal or minimal for a goal (predicate), meaning that F is maximal or minimal respectively for the set of propositions corresponding to the variables (arguments) of that goal (predicate).

Proposition 4.1. Let D be a database, G be a goal, and F be a dependency formula.

(a) *If F is in $gf_D(G)$, then for all substitutions σ, F is in $gf_D(G\sigma)$.*

(b) *Let F be maximal for G. Then F is in $gf_D(G)$ iff $G\theta_1 \cdots \theta_d$ is ground for all* SLDNF-*derivations for $D \cup \{G\}$ that succeed or flounder with mgu's $\theta_1, \ldots, \theta_d$.*

PROOF. Both results follow trivially from Definition 4.1. □

Thus, if a dependency formula F is maximal and is a groundness formula for some predicate, then that predicate always produces ground answers.

To apply Definition 4.1 to goals with evaluable predicates, Definition 2.7, describing when query evaluation flounders, must be extended. The new definition of "sufficiently instantiated" given in Definition 4.3 below accomplishes this. A table of some evaluable predicates from PROLOG, together with groundness formulae that represent their computational behavior, appears below:

Predicate	Groundness formula
true	*true*
$x_1 = x_2$	$g_1 \leftrightarrow g_2$
$x_1 >= x_2$	*true*
$maxint(x_1)$	g_1
$plus(x_1, x_2, x_3)$	$(g_2 \wedge g_3 \rightarrow g_1) \wedge (g_1 \wedge g_3 \rightarrow g_2) \wedge (g_1 \wedge g_2 \rightarrow g_3)$
x_1 *is* x_2	$g_2 \rightarrow g_1$

Groundness formulae for an evaluable predicate must correspond to the computational behavior of that predicate. Since the behavior of evaluable predicates is independent of any particular database, groundness formulae such as those given above would apply to every database.

As stated below, an evaluable literal is evaluated directly during query evaluation. Knowing groundness formulae for evaluable predicates permits the definition of "sufficiently instantiated" to be generalized to cover evaluable literals. This will ensure that, during query evaluation, an evaluable literal will be selected only when its groundness formula guarantees that selection and evaluation of that literal will contribute to the sequence of mgu's generated so far, that is, when its evaluation would ground some nonground term in the literal.

Definition 4.3. A literal is ***sufficiently instantiated*** if it is ground, or positive and not evaluable, or a positive, evaluable literal $p(t_1, \ldots, t_n)$ such that, given the groundness formula F for p and $S = \{i : t_i \text{ is ground}\}$, there exists some $j \notin S$, $1 \leq j \leq n$, such that $(\bigwedge_S g_i \wedge F \rightarrow g_j)$ is a tautology.

This extended definition means that a safe computation rule may delay selection of positive, evaluable literals. Properties of computation rules that delay the selection of evaluable predicates are discussed elsewhere [16]. While it is convenient to treat the equality predicate as an evaluable predicate for the purpose of the analysis described in the rest of this section, selection of equality predicates during SLDNF-resolution should not be delayed. Delaying the execution of equalities is unnecessary, because of their determinism, and undesirable, because it would delay the propagation of bindings.

Even given the above restriction on the selection of evaluable predicates, problems arise in the evaluation of some evaluable literals in some goals. Consider, for example, an evaluable predicate p of arity 4, with the groundness formula $(g_1 \rightarrow g_2) \wedge (g_3 \rightarrow g_4)$, and q of arity 2, with the groundness formula $g_1 \rightarrow g_2$. It could reasonably be expected that a goal such as $\leftarrow p(a, x_2, x_3, x_4) \wedge q(x_2, x_3)$ would propagate the binding from the first argument of p to the fourth. Only the first literal is sufficiently instantiated; evaluation of this literal is only certain to ground x_2. The second literal is then sufficiently instantiated, and evaluation of it is certain to ground x_3. If the first literal was not discarded by the query evaluation process, it could then be reevaluated to ground x_4. This means positive literals, of predicates with groundness conditions like that above, may have to be selected and partially evaluated at various points during SLDNF-resolution to achieve the expected behavior.

The groundness formula given above for p suggests that the arguments of p can be partitioned into two sets that are independent in terms of the propagation of bindings; this is an undesirable property. Rather than complicating the query evaluation process in this manner, it is therefore preferable to apply a further restriction on which dependency formulae may be groundness formulae for evaluable predicates: if a groundness formulae indicates that a currently nonground term in an evaluable literal would be ground if that literal were selected, then it must indicate that the whole literal would be ground if it were selected. In the absence of evaluable predicates with the same property as p, this is not a severe restriction. The restriction can be stated formally as follows.

Let the dependency formula F on P be the groundness formula for some evaluable predicate. Then, for all subsets P' of P, if there exists $g_j \in P \backslash P'$ such that $\bigwedge_{P'} g_i \wedge F \rightarrow g_j$, then for all $g_k \in P$, $\bigwedge_{P'} g_i \wedge F \rightarrow g_k$ is a tautology.

Note that the above restriction holds for all of the groundness formulae in the above table. As stated earlier, it is desirable to know one of the strongest groundness formulae, within the bounds of this restriction, for an evaluable predicate.

The construction of groundness formulae for goals can now be considered. The following proposition describes how a groundness formula can be determined for a predicate consisting of ground unit clauses.

Proposition 4.2. Let D be a database, p an extensional predicate in D, and F a maximal dependency formula for p. Then F is in $gf_D(p)$.

PROOF. For any n-ary extensional predicate p in D, a computation for $D \cup \{\leftarrow p(x_1, \ldots, x_n)\sigma\}$ will fail or succeed immediately, with any computed answers being ground substitutions. The result then follows from Definition 4.1. □

Note that Propositions 4.1 and 4.2 are consistent.

The following proposition describes how a groundness formula can be determined for a goal that is a single negative literal.

Proposition 4.3. Let D be a database, G be a goal $\leftarrow \sim p(t_1, \ldots, t_n)$, and F be in $gf_D(G)$. Then F is minimal.

PROOF. Let F be of the form $\bigwedge_i C_i \to g_i$, and $x_1, \ldots, x_m$ be the variables in G. Assume there exists C_j in F not equivalent to *false*. Let σ be a substitution for $x_1, \ldots, x_m$, and $S^\sigma = \{i : x_i\sigma \text{ is ground}, 1 \le i \le m\}$, such that $x_j\sigma$ is not ground and $\bigwedge_{S^\sigma} g_i \to C_j$ is a tautology. σ always exists, since it need only ground variables corresponding to the propositions in C_j, which never include x_j. Then $(\bigwedge_{S^\sigma} g_i \wedge F \to g_j)$ is a tautology, and from Definition 4.1, given mgu's $\theta_1, \ldots, \theta_d$ from a successful or floundering computation for $G\sigma$, $x_j\sigma\theta_1 \cdots \theta_d$ will be ground. But σ was chosen such that $x_i\sigma$, and therefore $G\sigma$, was not ground, so that $G\sigma$ will flounder immediately. Thus the initial assumption, that there exists C_j in F not equivalent to *false*, is false, and F must be minimal. □

The following proposition describes how a groundness formula can be determined for an atomic goal $\leftarrow p(t_1, \ldots, t_n)$ given a groundness formula for the n-ary predicate p.

Proposition 4.4. Let D be a database, G be a goal $\leftarrow p(t_1, \ldots, t_n)$, $S_j = \{i : x_i$ is a variable in $t_j\}$, F' be a dependency formula on $\{g'_1, \ldots, g'_n\}$ in $gf_D(p)$, and F be a dependency formula

$$\left(\left(g'_1 \leftrightarrow \left(\bigwedge_{S_1} g_i\right)\right) \wedge \cdots \wedge \left(g'_n \leftrightarrow \left(\bigwedge_{S_n} g_i\right)\right) \wedge F'\right) \Big\backslash \{g'_1, \ldots, g'_n\}$$

on the variables in G. Then F is in $\mathrm{gf_D(G)}$.

PROOF. Let $S^{t\sigma} = \{i : t_i\sigma \text{ is ground}\}$ and $S^{x\sigma} = \{i : x_i$ is a variable in G and $x_i\sigma$ is ground$\}$. Using Proposition 3.7(a), if $\bigwedge_{S^{x\sigma}} g_i \wedge F \to g_j$ is a tautology, then there is some g'_k such that $\bigwedge_{S^{t\sigma}} g'_i \wedge F' \to g_k$ is a tautology, where x_j is a variable in t_k. Since $t_k\sigma\theta_1 \cdots \theta_d$ is ground iff, for all $j \in S_k$, $x_j\sigma\theta_1 \cdots \theta_d$ is ground, the result follows from Definition 4.1. □

Using the notation of Section 3, the formula F above can be expressed more concisely and intuitively as

$$F'\left[g_1'\Big/\bigwedge_{S_1} g_i\right]\cdots\left[g_n'\Big/\bigwedge_{S_n} g_i\right].$$

Example 4.2. Using the groundness formula given for *append* in Example 4.1, the groundness formula derived for the atom.

$$append(x_1, x_2.x_3, 1.(2.(3.nil)))$$

is

$$((g_1' \leftrightarrow g_1) \wedge (g_2' \leftrightarrow g_2 \wedge g_3) \wedge (g_3' \leftrightarrow true) \wedge (g_1' \wedge g_2' \leftrightarrow g_3'))\backslash g_1'\backslash g_2'\backslash g_3',$$

or equivalently

$$(g_1' \wedge g_2' \leftrightarrow g_3')[g_1'/g_1][g_2'/g_2 \wedge g_3][g_3'/true],$$

which are equivalent to

$$g_1 \wedge g_2 \wedge g_3.$$

The resultant maximal dependency formula in Example 4.2 indicates that, as expected, evaluating the atom as a query would generate ground answers.

The next two propositions demonstrate how a groundness formula may be determined for a normal goal, given a groundness formula for each of the predicates that occurs as a literal in that goal.

Proposition 4.5. Let D be a database, and G be a goal $\leftarrow L_1 \wedge \cdots \wedge L_l$. Let L_i be a literal in G, and x_j a variable in L_i, such that for all computations for $L_i\sigma$ that succeed or flounder with mgu's $\theta_1,\ldots,\theta_d$, $x_j\sigma\theta_1\cdots\theta_d$ is ground. Then, for all computations for $G\sigma$ that succeed or flounder with mgu's $\eta_1,\ldots,\eta_e$, $x_j\sigma\eta_1\cdots\eta_e$ is ground.

PROOF. It is a straightforward inductive proof on the length of refutation to show that given any SLDNF-derivation $G\sigma$ that succeeds or flounders with mgu's $\eta_1,\ldots,\eta_e$, there is a corresponding SLDNF-derivation for $L_i\sigma$ that succeeds or flounders with mgu's $\theta_1,\ldots,\theta_d$, such that $\sigma\theta_1\cdots\theta_d$ is more general than $\sigma\eta_1\cdots\eta_e$. The result then follows. □

Proposition 4.6. Let D be a database, G be a goal $\leftarrow L_1 \wedge \cdots \wedge L_l$, and F_k be in $gf_D(\leftarrow L_k)$, $1 \le k \le l$. Then $F_1 \wedge \cdots \wedge F_l$ is in $gf_D(G)$.

PROOF. Let $x_1,\ldots,x_n$ be the variables in G, $V_k = \{i : x_i \text{ is a variable in } L_k\}$, and $S_1 = \{g_i : x_i\sigma \text{ is ground}\}$, where σ is a substitution for $x_1,\ldots,x_n$. Then, from Proposition 3.8, if $\bigwedge_{S_1} g_i \wedge (F_1 \wedge \cdots \wedge F_l) \rightarrow g_j$ is a tautology, then there exists a sequence of sets $S_1,\ldots,S_{s+1}$, $s \ge 0$, where $j \in S_{s+1}$ with the following property. For all t, $1 \le t \le s$, there exists some g_k, $1 \le k \le n$, $k \notin S_t$, and some F_m, $1 \le m \le l$, such that $S_{t+1} = S_t \cup \{k\}$ and $\bigwedge_{S_t} g_i \wedge F_m \rightarrow g_k$ is a tautology. Now, if $s = 0$, then $j \in S_1$ and $x_j\sigma$ is ground. If $s = 1$, then from Definition 4.1 and Proposition 4.5, $j \in S_2$ and $x_j\sigma\theta_1\cdots\theta_d$ is ground for all successful or floundering computations for $G\sigma$. Consider a sequence with $s > 1$. Then $x_k\sigma\theta_1\cdots\theta_d$ is ground for all successful or floundering computations for $G\sigma$, for every k added to S_1 to get S_s. Since there is some F_m such that $\bigwedge_{S_s} g_i \wedge F_m \rightarrow g_j$ is a tautology, then the result holds from Definition 4.1 and Proposition 4.5. □

Example 4.3. Using the groundness formula given for *append* in Example 4.1 and given the groundness formula $g_1 \leftrightarrow g_2$ for $=$, a groundness formula for the goal $\leftarrow append(x_1, x_2, x_3) \wedge x_1 = x_3$ is

$$(g_1 \wedge g_2 \leftrightarrow g_3) \wedge (g_1 \leftrightarrow g_3),$$

which is equivalent to

$$(g_1 \rightarrow g_2 \wedge g_3) \wedge (g_3 \rightarrow g_1 \wedge g_2).$$

Note that the groundness formula derived for the goal

$$\leftarrow (x_1' = t_1) \wedge \cdots \wedge (x_n' = t_n) \wedge p(x_1', \ldots, x_n')$$

via Proposition 4.6 is consistent with the groundness formula derived for $\leftarrow p(t_1, \ldots, t_n)$.

Proposition 4.7 below demonstrates how a groundness formula may be derived for a predicate p if a groundness formula is known for each of the predicates that occur in the body of each of the clauses. Where the set of clauses defining a predicate is to be analysed to determine some property of the predicate, the analysis may be simplified by considering a modified set of clauses in which the head of each clause contains only variables [25].

Definition 4.4. The *homogeneous* form of a clause C of the form

$$p(t_1, \ldots, t_n) \leftarrow G$$

is a clause of the form

$$p(x_1, \ldots, x_n) \leftarrow (x_1 = t_1) \wedge \cdots \wedge (x_n = t_n) \wedge G,$$

where $x_1, \ldots, x_n$ are distinct and do not occur in C. Let D be a database and p be an n-ary predicate defined in D. Then the *homogeneous* form of p is the set of homogeneous forms of the clauses defining p in D.

Example 4.4. The following is the standard definition of the ternary predicate *append*, used for concatenating or splitting lists:

$$append(nil, y_1, y_1) \leftarrow$$

$$append(y_2.y_3, y_4, y_2.y_5) \leftarrow append(y_3, y_4, y_5).$$

The homogeneous form of *append* is

$$append(x_1, x_2, x_3) \leftarrow (x_1 = nil) \wedge (x_2 = y_1) \wedge (x_3 = y_1)$$

$$append(x_1, x_2, x_3) \leftarrow (x_1 = y_2.y_3) \wedge (x_2 = y_4) \wedge (x_3 = y_2.y_5) \wedge append(y_3, y_4, y_5).$$

Proposition 4.7. Let D be a database and p an n-ary predicate in D. Let the homogeneous form of p be

$$A \leftarrow B_1$$
$$\vdots$$
$$A \leftarrow B_l.$$

Let $V_k = \{g_i : x_i$ *is a variable in* B_k *not in* $A\}$, *and* F_m *be in* $gf_D(B_m)$, $1 \le m \le l$. *Then* $(F_1 \backslash V_1) \dot{\vee} \cdots \dot{\vee} (F_l \backslash V_l)$ *is in* $gf_D(p)$.

PROOF. Since the homogeneous form of p is computationally equivalent to the definition of p, only the homogeneous form of p need be considered. Let $S^\sigma = \{i : x_i$ in A and $x_i\sigma$ is ground$\}$. By Proposition 3.9, if

$$\left(\bigwedge_S g_i \wedge (F_1 \backslash V_1) \dot{\vee} \cdots \dot{\vee} (F_l \backslash V_l) \rightarrow g_j \right) \text{ is a tautology,}$$

then, for all m, $1 \le m \le l$,

$$\left(\bigwedge_S g_i \wedge (F_m \backslash V_m) \rightarrow g_j \right) \text{ is a tautology.}$$

Then by Proposition 3.7(a) and Definition 4.1, for a computation for $p(x_1, \ldots, x_n)\sigma$ that succeeds or flounders with mgu's $\theta_1, \ldots, \theta_d$, $x_j\sigma\theta_1 \cdots \theta_d$ will be ground no matter which clause of p is the first input clause to the computation. The result then follows from Definition 4.1. □

Note that Propositions 4.6 and 4.7 are consistent with Proposition 4.2.

Example 4.5. The following definition is for a predicate that holds if the first argument is a singleton list of the first element in the second argument list, or both arguments are nil:

$$p(x_1, x_2) \leftarrow x_1 = nil \wedge x_2 = nil$$
$$p(x_1, x_2) \leftarrow x_1 = x_3.nil \wedge x_2 = x_3.x_4.$$

Given the groundness formula $g_1 \leftrightarrow g_2$ for $=$, a groundness formula for p is

$$(((g_1 \leftrightarrow true) \dot{\wedge} (g_2 \leftrightarrow true)) \backslash \{\,\}) \dot{\vee} (((g_1 \leftrightarrow g_3) \dot{\wedge} (g_2 \leftrightarrow g_3 \wedge g_4)) \backslash \{g_3, g_4\}),$$

which simplifies to $g_2 \rightarrow g_1$.

Example 4.6. Given the definition

$$q(x_1, x_2, x_3) \leftarrow x_1 \ge x_2 \wedge (x_4 \; is \; x_1 - x_2) \wedge r(x_4, x_5) \wedge (x_3 \; is \; x_1 + x_5)$$
$$q(x_1, x_2, x_3) \leftarrow x_1 < x_2 \wedge (x_4 \; is \; x_2 - x_1) \wedge r(x_4, x_5) \wedge (x_3 \; is \; x_2 + x_5)$$

and given the groundness formula $g_2 \rightarrow g_1$ for *is*, a groundness formula for the first clause of q is

$$(true \dot{\wedge} (g_1 \wedge g_2 \rightarrow g_4) \dot{\wedge} (g_4 \wedge g_5) \dot{\wedge} (g_1 \wedge g_5 \rightarrow g_3)) \backslash \{g_4, g_5\},$$

which simplifies to $g_1 \rightarrow g_3$. Similarly, a groundness formula for the second clause is $g_2 \rightarrow g_3$. A groundness formula for q can then be derived as $(g_1 \rightarrow g_3) \dot{\vee} (g_2 \rightarrow g_3)$, which is equivalent to $g_1 \wedge g_2 \rightarrow g_3$.

Groundness Formulae for Recursive Databases

Using the definitions and propositions given so far, groundness formulae can be derived for the predicates of a hierarchical database given groundness formulae for the evaluable predicates. The following definition and two propositions provide extensions to allow groundness formulae to be derived for the predicates defined in a recursive database.

Definition 4.5. Let D be a database, G be a goal with variables $x_1, \ldots, x_n$, and $S^\eta = \{i : x_i\eta \text{ is ground}\}$, where η is a substitution for $x_1, \ldots, x_n$. Then a dependency formula F on $\{g_1, \ldots, g_n\}$ is a *groundness formula* to length r for G with respect to D if the following holds: For all substitutions σ and for all j, $1 \leq j \leq n$, if

$$\bigwedge_{S^\sigma} g_i \wedge F \rightarrow g_j$$

is a tautology, then for all SLDNF-derivations of length $\leq r$ for $D \cup \{G\sigma\}$ that succeed or flounder with mgu's $\theta_1, \ldots, \theta_d$,

$$x_j\sigma\theta_1 \cdots \theta_d$$

is ground.

$gf_D^r(G)$ denotes the set of all groundness formulae to length r for G with respect to D.

Definition 4.6. Let D be a database, p be an n-ary predicate in D, and F be a dependency formula. Then F is a groundness formula to length r for p with respect to D, if F is in $gf_D^r(\leftarrow p(x_1, \ldots, x_n))$.

$gf_D^r(p)$ denotes the set of all groundness formulae to length r for p with respect to D.

From Definition 4.5 it follows that if F is in $gf_D^{r+1}(G)$, then F is in $gf_D^r(G)$. From Definitions 4.1 and 4.5 it follows that $gf_D^\infty(G)$ and $gf_D(G)$ are equivalent.

Propositions 4.8, 4.9, and 4.10 correspond to Propositions 4.4, 4.6, and 4.7 respectively, extended to take the limit on derivation length into account.

Proposition 4.8. Let D be a database, G be a goal $\leftarrow p(t_1, \ldots, t_n)$, $S_j = \{i : x_i$ is a variable in $t_j\}$, F' be a dependency formula on $\{g'_1, \ldots, g'_n\}$ in $gf_D^r(p)$, and F be a dependency formula

$$\left(\left(g'_1 \leftrightarrow \left(\bigwedge_{S_1} g_i\right)\right) \wedge \cdots \wedge \left(g'_n \leftrightarrow \left(\bigwedge_{S_n} g_i\right)\right) \wedge F'\right) \Big\backslash \{g'_1, \ldots, g'_n\}$$

on the variables in G. Then F is in $gf_D^r(G)$.

PROOF. The proof of Proposition 4.4 can be extended trivially to prove this result. □

Proposition 4.9. Let D be a database, G be a goal $\leftarrow L_1 \wedge \cdots \wedge L_l$, and F_k^r be in $gf_D^r(\leftarrow L_k)$, $1 \le k \le l$. Then $F_1^r \dot{\wedge} \cdots \dot{\wedge} F_l^r$ is in $gf_D^r(G)$.

PROOF. Proposition 4.6 shows that $F_1^\infty \dot{\wedge} \cdots \dot{\wedge} F_l^\infty$ is in $gf_D^\infty(G)$ and hence in $gf_D(G)$. The corresponding proof is extended by adding that a refutation of length r for $D \cup \{G\sigma_1\}$ cannot include a refutation of length greater than r for $D \cup \{L_i\sigma_2\}$, for any L_i, $1 \le i \le l$. □

Proposition 4.10. Let D be a database, and p be an n-ary predicate in D. Let the homogeneous form of p be

$$A \leftarrow B_1$$
$$\vdots$$
$$A \leftarrow B_l$$

Let $V_k = \{g_i : x_i$ is a variable in B_k not in $A\}$, and F_m^r be in $gf_D^r(B_m)$, $1 \le m \le l$. Then $(F_1^r \setminus V_1) \dot{\vee} \cdots \dot{\vee} (F_l^r \setminus V_l)$ is in $gf_D^{r+1}(p)$.

PROOF. Proposition 4.7 shows that $(F_1^\infty \setminus V_1) \dot{\vee} \cdots \dot{\vee} (F_l^\infty \setminus V_l)$ is in $gf_D(p)$. The corresponding proof is extended by adding that a refutation of length $r+1$ for $D \cup \{\leftarrow p(x_1, \ldots, x_n)\sigma_1\}$ cannot include a refutation of length greater than r for $D \cup \{B_m\sigma_2\}$ for any B_i, $1 \le m \le l$. □

The following algorithm and proposition demonstrate how groundness formulae can be constructed for each predicate in a recursive normal database.

Groundness Formula Algorithm.

Input: A database D.

Output: A set Γ_D of pairs $\langle p, F_p \rangle$, where p is a predicate and F_p is a dependency formula.

Method:

```
$r := 0$;
**for each** predicate $p$ in $D$ **do**
  **if** $p$ is evaluable **then let** $F_p^r$ be the groundness formula for $p$
  **else** let $F_p^r$ be any maximal dependency formula for $p$;
**repeat**
  $r := r + 1$;
  **for each** predicate $p$ in $D$ **do**
    **if** $p$ is evaluable **then let** $F_p^r$ be the groundness formula for $p$
    **else** let $F_p^r$ be constructed by Propositions 4.8, 4.9, and 4.10 from the
        dependency formula $F_q^{r-1}$ for each of the predicates $q$ for which
        atoms occur in the bodies of the clauses defining $p$ in $D$;
**until** $F_p^r$ is logically equivalent to $F_p^{r-1}$ for each predicate $p$ in $D$;
**return** $\Gamma_D = \{\langle p, F_p^r \rangle \mid p$ is a predicate in $D\}$;
```

Proposition 4.11. Let D be a database. Then

(a) *The groundness formula algorithm with D as input terminates;*

(b) *for each $\langle p, F_p \rangle \in \Gamma_D$, F_p is in $gf_D(p)$.*

PROOF. Let p be a predicate in D. If p is evaluable, $F_p^{r-1} = F_p^r$ for $r > 0$. If p is not evaluable, F_p^0 is maximal for p. Therefore $F_p^0 \to F_p^1$ is a tautology. If p is not evaluable, the derivation of F_p^r has the form

$$\left(F_{q_1}^{r-1} \dot{\wedge} \cdots \dot{\wedge} F_{q_n}^{r-1}\right) \backslash V_1 \dot{\vee} \cdots \dot{\vee} \left(F_{q_m}^{r-1} \dot{\wedge} \cdots \dot{\wedge} F_{q_l}^{r-1}\right) \backslash V_p,$$

where each V_j is a set of propositions, for $r > 0$. Since $F_1 \to F_1'$ implies that $F_1 \dot{\wedge} F_2 \to F_1' \dot{\wedge} F_2$ (by Proposition 3.3), $F_1 \dot{\vee} F_2 \to F_1' \dot{\vee} F_2$ (by Proposition 3.5), and $F_1 \backslash g_k \to F_1' \backslash g_k$ [by Proposition 3.7(c)], then $F_p^{r-1} \to F_p^r$ must hold. Definition 3.2 defines a partial order on the set of dependency formulae for logical implication. Since $F_p^{i-1} \to F_p^i$ for all predicates p in D, there exists $f > 0$ such that for all predicates p in D, F_p^f is logically equivalent to F_p^{f-1}, and (a) holds. It follows from Propositions 4.8, 4.9, and 4.10 that F_p^r is a $gf_D^r(p)$. It is trivially true that for any F_p^r constructed for $r > f$, F_p^r is logically equivalent to F_p^f. Therefore F_p^f is logically equivalent to F_p^∞ and (b) holds. □

Example 4.7. Let D be the database consisting of the predicate *append* from Example 4.4 and the evaluable predicate $=$. Applying the algorithm described above, given the groundness formula $g_1 \leftrightarrow g_2$ for $=$, gives the following results:

r	F_{append}^r	$F_=^r$
0	$g_1 \wedge g_2 \wedge g_3$	$g_1 \leftrightarrow g_2$
1	$g_1 \wedge (g_2 \leftrightarrow g_3)$	$g_1 \leftrightarrow g_2$
2	$g_1 \wedge g_2 \leftrightarrow g_3$	$g_1 \leftrightarrow g_2$
3	$g_1 \wedge g_2 \leftrightarrow g_3$	$g_1 \leftrightarrow g_2$

Since F_{append}^3 is equivalent to F_{append}^2 and $F_=^3$ is equivalent to $F_=^2$, the algorithm terminates, returning $\Gamma_D = \{\langle append, g_1 \wedge g_2 \leftrightarrow g_3 \rangle, \langle =, g_1 \leftrightarrow g_2 \rangle\}$.

Note that the groundness formula derived for *append* in Example 4.7 is the same as that given in Example 4.1.

5. CONNECTED DATABASES

This section defines the class of *connected databases*. A result describing desirable properties of connected databases, similar to that given in Section 2 for allowed databases, is then proven.

Definition 5.1. Let D be a normal database and G be a normal goal. A database clause in D with homogeneous form $p(x_1, \ldots, x_n) \leftarrow B$ is *connected* if, for some F in $gf_D(\leftarrow B)$, $g_1 \wedge \cdots \wedge g_n \wedge F$ is maximal for $\leftarrow B$. D is *connected* if each clause in D is connected. $D \cup \{G\}$ is *connected* if D is connected and, for some F in $gf_D(G)$, F is maximal for G.

Given this definition, connectedness is a semantic property, based on the behavior of databases and goals under SLDNF-resolution. The algorithm for deriving groundness formulae given in Section 4 uses the syntactic form of a database. The groundness formulae derived for a database can therefore be used to provide a syntactic check of whether the database is connected.[4] Having derived groundness formulae for a given database D, a groundness formula can be derived for any goal G with respect to D, and therefore, whether G is connected with respect to D can be tested without reconstructing groundness formulae for D. If the groundness formulae constructed for a database by the algorithm do not meet the requirements of Definition 5.1, no information about the connectedness of the database is revealed. Note that by Definition 5.1, if every variable that occurs in a clause occurs in its head, then that clause is connected.

Example 5.1. Let D be the following database:

$$p(x,y) \leftarrow r(x) \wedge \sim s(y)$$

$$q(x,y) \leftarrow \sim s(x) \wedge t(y),$$

where the unary predicates r, s, and t are extensional predicates. Let G be the goal $\leftarrow p(x,y) \wedge q(x,y)$. Using the construction described in Section 4,

$$\Gamma_D = \{\langle p, g_1\rangle, \langle q, g_2\rangle, \langle r, g_1\rangle, \langle s, g_1\rangle, \langle t, g_1\rangle\}.$$

Using the groundness formulae in Γ_D, we have that D, G, and $D \cup \{G\}$ are connected.

Note that, in Example 5.1, both of the clauses are admissible but not allowed, G is allowed, and $D \cup \{G\}$ is not allowed.

Some desirable properties of connected databases are now considered.

Theorem 5.1. Let D be a connected normal database and G be a normal goal. If $D \cup \{G\}$ is connected and R is a safe computation rule, then the following properties hold:

(a) *Every R-computed answer for $D \cup \{G\}$ is a ground substitution for all free variables in G.*

(b) *Query evaluation of $D \cup \{G\}$ via R does not flounder.*

PROOF. (a): This result follows directly from Definition 5.1 and Proposition 4.1.

(b): We begin by proving inductively that, for every goal G_i in an SLDNF-derivation $G_0 = G, G_1, \ldots$ of $D \cup \{G\}$ via R, $D \cup \{G_i\}$ is connected. For $i = 0$, the result is trivially true. Assume the result is true for G_0 to G_i. Let L be the literal selected by R from G_i. If L is negative, the result holds, as $G_{i+1} \subset G_i$ and any groundness formula for L is minimal. Let L be $p(t_1, \ldots, t_n)$. If p is evaluable, then

[4] In fact, an equivalent condition for a database D to be connected can be applied as part of the algorithm for deriving groundness formulae. D is connected if during the construction of F_p^r, for all p and r (up to the fixpoint), every application of the exclusion operation having the form $F \backslash g_i$ has the C_i corresponding to g_i *not* equivalent to *false*. Informally, this corresponds to every local variable having a satisfiable condition under which it will be ground.

let θ be the substitution generated by evaluating L. Then $L\theta$ will be ground, and all of the variables in L that occur in G_i will be ground in G_{i+1}, and the result holds. If p is not evaluable, then, for any clause of p selected, that clause is connected, and by Definition 5.1 the result holds. Therefore $D \cup \{G_i\}$ is connected for all $i \geq 0$.

Consider a goal G for which $D \cup \{G\}$ is connected. Assume G contains no sufficiently instantiated literals. Then G consists only of (nonground) negative and evaluable literals. A goal consisting of only negative literals must be ground to be connected; therefore G contains at least one nonground evaluable literal. Consider each literal L in G. If L is negative and F is in $gf_D(\leftarrow L_j)$, then F is minimal. If L is evaluable, let L be of the form $p(t_1, \ldots, t_n)$, F be a groundness formula for p, and $S = \{i : t_i \text{ is ground}\}$. By Definition 4.3 and the restriction on dependency formulae for evaluable predicates, there is no $g_j \notin S$, $1 \leq j \leq n$, such that $\wedge_S g_i \wedge F \rightarrow g_j$ is a tautology, since L is not sufficiently instantiated. Then if x_k is a variable in an evaluable literal in G, and F is in $gf_D(G)$, then using Proposition 3.8, it cannot hold that $F \rightarrow g_k$ is a tautology. Also, F maximal for G is a $gf_D(G)$, since $D \cup \{G\}$ is connected; therefore $F \rightarrow g_k$ is a tautology. Thus the assumption that G contains no sufficiently instantiated literals as false, and (b) must hold. □

The following theorem shows that the class of allowed databases is a subclass of the class of connected databases.

Theorem 5.2. Let $D \cup \{G\}$ be allowed. Then $D \cup \{G\}$ is connected.

PROOF. It is sufficient to prove that, if p is a predicate in D defined only by allowed clauses, then every maximal dependency formula for p is in $gf_D(p)$. This can be shown directly from Proposition 4.1 and the theorem in Section 2 describing the query evaluation properties of allowed databases and goals. Alternatively, this property can be demonstrated (more usefully) by observing that the groundness formula derived for p by constructing Γ_D will be maximal for p. This will hold even if the groundness formulae derived for predicates with definitions that include nonallowed clauses are all minimal. The result can then be shown to hold by using a maximal dependency formula as a groundness formula for predicates in D defined only by allowed clauses, and minimal dependency formula for all other predicates in D, to satisfy the sufficient condition described by Definition 5.1. □

Example 5.1 demonstrates that the converse of Theorem 5.2 is false.

6. CONCLUSION

A new class of deductive databases, connected databases, has been defined. It is possible to identify goals that will not flounder and will produce ground answers on this class of database. The class of connected databases is much less restrictive than the class of allowed databases, permitting the inclusion of system predicates such as *plus* and general equality, and a greater class of recursive definitions such as for *append*, *member*, and *reverse*.

Various methods can be applied to improve the analysis described in Section 4. For example, consider the following database D:

$$p(x_1) \leftarrow f(x_1, 1) = f(x_2, x_2).$$

The groundness formula for p derived by the algorithm of Section 4 is $(g_1 \leftrightarrow g_2) \backslash g_2$, which is equivalent to *true*. This groundness formula cannot be used to show that $D \cup \{\leftarrow p(x)\}$ is connected. After transforming D to the database D' by

$$p(x_1) \leftarrow x_1 = x_2 \wedge 1 = x_2,$$

the groundness formula for p is $((g_1 \leftrightarrow g_2) \wedge g_2) \backslash g_2$, which is equivalent to g_1. Then $D' \cup \{\leftarrow p(x)\}$ is connected. In general, an equality can be partially interpreted, according to the equality axioms [14], to reduce it from the form $t = t'$ to the form $v_1 = t_1 \wedge \cdots \wedge v_n = t_n$ where every t is a term and every v is a variable. Applying the equality axioms in this way allows the algorithm in Section 4 to analyse the dependencies between variables more precisely.

Unfolding [19, 10] can also be used to achieve a more precise analysis of dependencies. Dependency analysis is performed at the level of variables within the body of a clause, but is limited by the arguments to predicates in passing dependencies from one predicate to another. Unfolding can, in some cases, eliminate the loss of resolution involved in considering dependencies in terms of arguments to a predicate. For example, the predicate p in the database D given by

$$p(f(x_1, x_2)) \leftarrow x_1 = x_2$$

has the groundness formula *true*, which cannot be used to show that $D \cup \{\leftarrow p(f(1, x))\}$ is connected. Applying unfolding gives $D \cup \{\leftarrow 1 = x\}$, which is trivially connected.

The algorithm described in Section 4 has been implemented in NU-PROLOG [20]. Many improvements on the basic algorithm were implemented; in particular, the groundness formula for a predicate is only recalculated if a new groundness formula was derived for a literal in a body of that predicate, and equalities are partially evaluated as described above. The implementation operates in two phases. The first computes groundness formulae for a database and reports where the database is not connected. The second is an adaptation of NU-PROLOG query language that transforms queries, using a modified Lloyd-Topor transformation [12], and checks that they are connected before passing them to the interpreter.

The implementation was used successfully in 1986 for a "Database Systems" course in which students use the NU-PROLOG query language to interrogate databases. The addition of groundness formula checking to the interface does not significantly degrade the response time, since only the query (and the groundness formulae derived for the database) need be examined.

Dependency formulae have been used to represent type dependencies [5], and further research is being conducted into the use of dependency formulae to represent functional and finiteness dependencies, and to apply dependency analysis to tasks such as query optimization through subquery reordering.

I would like to thank my supervisors, Rodney Topor and John Lloyd. I would also like to thank Justin Zobel for assistance with this paper, and Jeffrey Schultz for assistance with NU-PROLOG. Thanks are also due to James Thom and to the anonymous referees for their detailed comments and suggestions. This work was done while at the Department of Computer Science, The Univeristy of Melbourne, Australia.

REFERENCES

1. Bruynooghe, M. and Jenssens, G., An Instance of Abstract Interpretation Integrating Type and Mode Inferencing (Extended Abstract), in: *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Wash., Aug. 1988, pp. 669–683.
2. Chang, C. C. and Keisler, H. J., *Model Theory*, North-Holland, Amsterdam, 1973.
3. Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Databases*, Plenum, 1978, pp. 293–322.
4. Codd, E. F., Relational Completeness of Data Base Sublanguages, in: R. Rustin (ed.), *Data Base Systems*, Prentice-Hall, 1972, pp. 65–98.
5. Dart, P. W. and Zobel J., Transforming Typed Logic Programs to Well-Typed Logic Programs, Technical Report 88/11, Dept. of Computer Science, Univ. of Melbourne, 1988.
6. Debray, S. K. and Warren, D. S., Automatic Mode Inference for Prolog Programs, in: *Proceedings of the 1986 Symposium on Logic Programming*, Salt Lake City, Sept. 1986, pp. 78–88.
7. Debray, S. K., Efficient Dataflow Analysis of Logic Programs, in: *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, Calif., Jan. 1988, pp. 260–273.
8. Decker, H., Integrity Enforcement in Deductive Databases, in: *Proceedings of the First International Conference on Expert Database Systems*, Charleston, S.C., 1986.
9. Demolombe, R., *Syntactical Characterization of a Subset of Domain Independent Formulas*, ONERA-CERT, Toulouse, 1982.
10. Kanamori, T. and Horiuchi, K., Construction of Logic Programs Based on Generalised Fold/Unfold Rules, in: *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pp. 744–768.
11. Kifer, M., Ramakrishnan, R., and Silberschatz, A., An Axiomatic Approach to Deciding Query Safety in Deductive Databases, in: *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Austin, Tex., Mar. 1988, pp. 52–60.
12. Lloyd, J. W. and Topor, R. W., Making Prolog More Expressive, *J. Logic Programming* 1(3):225–240 (1984).
13. Lloyd, J. W. and Topor, R. W., A Basis for Deductive Database Systems. II, *J. Logic Programming* 3(1):55–67 (Apr. 1986).
14. Lloyd, J. W., *Foundations of Logic Programming*, 2nd ed., Springer-Verlag, New York, 1987.
15. Mellish, C. S., Automatic Generation of Mode Declarations in Prolog Programs, in: *Proceedings of the Workshop on Logic Programming for Intelligent Systems*, Long Beach, Calif., Sept. 1981.
16. Naish, L., *Negation and Control in Prolog*, Springer-Verlag, New York, 1986.
17. Nicolas, J., Logic for Improving Integrity Checking in Relational Data Bases, *Acta Inform.* 18:227–253 (1982).

18. Somogyi, Z., A System of Precise Modes for Logic Programs, in: *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pp. 769–787.
19. Tamaki, H. and Sato, T., Unfold/Fold Transformation of Logic Programs, in: *Proceedings of the Second International Logic Programming Conference*, Uppsala, Sweden, July 1984, pp. 127–138.
20. Thom, J. A. and Zobel, J. (eds.), NU-Prolog Reference Manual, Version 1.3, Technical Report 86/10, Dept. of Computer Science, Univ. of Melbourne, 1986; revised, May 1988.
21. Topor, R. W., Domain-Independent Formulas and Databases, *Theoret. Comput. Sci.* 52(3):281–306 (1987).
22. Topor, R. W. and Sonenberg, E. A., On Domain Independent Databases, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, Calif., 1988, pp. 217–240.
23. Ullman, J. D., *Principles of Database Systems*, Pitman, London, 1982.
24. Ullman, J. D., *Principles of Database Knowledge-Base Systems*, Vol. I, Computer Science Press, Rockville, Md., 1988.
25. van Emden, M. H. and Lloyd, J. W., A Logical Reconstruction of Prolog II, *J. Logic Programming* 1(2):143–149 (Aug. 1984).
26. Warren, D. H. D., Implementing Prolog—Compiling Predicate Logic Programs, Research Reports 39 and 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, May 1977.
27. Warren, R., Hermenegildo M., and Debray, S. K., On the Practicality of Global Flow Analysis of Logic Programs, in: *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, Seattle, Wash., Aug. 1988, pp. 684–699.
28. Zaniolo, C., Safety and Computation of Non-recursive Horn Clauses, in: *Proceedings of the Second International Conference on Expert Database Systems*, Charleston, S.C., Apr. 1986, pp. 167–178.