EFFICIENT BOTTOM-UP COMPUTATION
OF QUERIES ON STRATIFIED DATABASES

I. BALBIN, G. S. PORT*, K. RAMAMOHANARAO*, AND K. MEENAKSHI*

Magic-set transformations on (function-free) deductive databases that contain only positive body literals permit the efficient bottom-up evaluation of answers to a wide class of queries. We investigate the applicability of magic sets to stratified and allowed databases containing negative body literals. We introduce a new, less restrictive definition of allowedness. We then present an algorithm that performs a magic-set-based transformation on an initially stratified and allowed database. The answers to a query on the transformed database are evaluated using a structured bottom-up computation which retains the efficiency of the magic-set approach. The set of answers is then proved sound and complete with respect to the perfect model.

1. INTRODUCTION

“What use is magic if it can’t save a unicorn?”
—Peter S. Beagle, The Last Unicorn.

One area in the field of logic programming, known as deductive databases [17], is concerned with developing logic-based programming systems which manipulate large quantities of data efficiently. In terms of logic programming, deductive databases are just logic programs without function symbols. Logic programs are usually computed top-down (e.g. PROLOG). For deductive databases, it is a matter of current research whether top-down, bottom-up, or hybrid forms of computation are most suitable for the manipulation of large quantities of data. At
an abstract level, mathematical logic provides a uniform framework for the expression and manipulation of data. A major strength from the point of view of computer science is that the manipulation of data can be given a clear and precise model-based semantics.

Techniques have already been developed for relational database systems to manipulate large amounts of data very efficiently. The way data are handled in these systems can be expressed within a subset of logic. These systems typically allow the retrieved data to be transformed using a fixed set of operations, but fall short of providing a general computational mechanism for transforming data; for example, it is not possible to express the transitive closure of a relation in a traditional database system. Deductive databases extend the expressive power of relational database systems by adding recursion.

In inferring answers to a query on a database, a bottom-up computation can naturally employ the existing optimization techniques developed for relational databases. As a consequence, bottom-up computation is the focus of much research into deductive databases [8,12,6,40,33,38,23]. However, bottom-up computation does not naturally make use of ground terms in a literal in the same "goal-driven" way that a top-down computation does. A direct consequence is that many irrelevant tuples may be generated during a bottom-up computation. Research is currently concentrated on minimizing the generation of irrelevant tuples.

A commonly accepted approach seeks to perform a compile-time transformation of the database, based on the query, into an equivalent form which enables a bottom-up computation to focus on relevant tuples. Two examples of this approach are magic sets [8,12,39] and the less general counting sets [8,40,18]. In this paper, we focus on the use of magic-set transformations on (function-free) deductive databases that include negative body literals. The magic set algorithms on these databases are based on a sideways information-passing strategy\(^1\) (SIPS).

Two common properties that databases should have which enable answers to queries to be answered both correctly and efficiently are stratification and domain independence. Assuming the classical two-valued logic, there is currently no convenient semantics for unstratified databases, and so we require that the initial database be stratified [14]. The perfect model semantics [1] was defined for the class of stratified databases, and we take this to be the semantics of our databases. Domain independence is an essential property of a database because it ensures that the semantics of the database does not change when new constants are added to its language. Allowedness is a sufficient syntactic characterization of domain independence [45]. In addition, it leads to certain efficiencies during the computation, and so we focus here on only allowed databases.

When the generalized magic-sets (GMS) transformation [12] is applied to a stratified data base, the resultant database may be unstratified. Unlike [11], we claim that handling an unstratified database does "pose a problem" and requires a different approach to database transformation and underlying bottom-up computation. We propose a new method that retains the efficiency of magic sets by using a structured bottom-up computation.

\(^1\)Sometimes we will mean the plural strategies. There should be no confusion, and we retain the same acronym.
The paper is organized as follows. Notation and preliminary definitions are presented in the rest of this section. In Section 2, we briefly describe and contrast the bottom-up computation of a query to a deductive database which does not contain any negative body literals (positive database) and for a database which does contain negative body literals (normal database). In Section 3, we review the notion of SIPS and show how they may be used to adorn a database. In Section 4, we present an improved and simplified magic-set algorithm which can be used to implement given SIPS. We show that the magic-set algorithm preserves a form of uniform equivalence \[42\] with respect to an initial positive database. In Section 5, we discuss the effect of the magic-set algorithm on normal deductive databases. We present a new definition of allowedness, called allowedness with respect to a query, and show that for allowed SIPS the magic-set algorithm preserves allowedness. In Section 6, the factors that contribute to the unstratification of the database are analysed, and we show that for the magic sets constructed for positive literals, the cause of unstratification is removed by employing a labeling algorithm to be performed prior to the magic-set algorithm. The labeling algorithm alone does not guarantee stratification when we take into account magic sets constructed for negative literals. In Section 7, we build on the intuition of labeling and present a new structured bottom-up method for magic sets. The method employs the magic sets constructed for negative literals without the need for explicit labeling and preserves the perfect model semantics of the database. We then present a summary of our system indicating some further possible improvements. Section 8 describes related work, after which we make some concluding remarks.

1.1. Terminology

The language of a deductive database consists of the variables, constants, and predicate names in the database. We adopt the convention of denoting variables by identifiers beginning with an uppercase letter and constants by identifiers beginning with a lowercase letter. In the absence of function symbols, a term is either a constant or a variable. Identifiers starting with lowercase letters are used for predicate names.

An atom is written as \( p(t_1, t_2, \ldots, t_n) \), \( n \geq 0 \), where \( p \) is a predicate name and \( t_1, t_2, \ldots, t_n \) are terms (parentheses are dropped when \( n = 0 \)). A literal is either an atom, or an atom preceded by the negation sign \( \neg \).

A rule is a statement of the form

\[ p_0 \leftarrow p_1, p_2, \ldots, p_m, \quad m \geq 0, \]

where the atom \( p_0 \) is known as the head, and \( p_1, \ldots, p_m \) are distinct literals. The conjugation \( p_1, p_2, \ldots, p_m \) is called the body, and each \( p_i \) is a body literal. When \( m = 0 \), the rule is written as the atom \( p_0 \) and is also known as a unit clause. Without loss of generality, a query is a statement of the form \( q \leftarrow q_1, \ldots, q_m \) and asking \( \leftarrow q \).

An atom \( p(t_1, t_2, \ldots, t_n) \), \( n \geq 0 \), is ground when all of its terms, \( t_1, t_2, \ldots, t_n \), are constants. A ground rule is one in which each atom in the rule is ground. A fact is a ground unit clause. The definition of a predicate \( p \) is the set of rules which have \( p \) as the head predicate. A base predicate is defined solely by facts. The set of facts
in the database is also known as the extensional database. A rule that is not a fact is known as a derivation rule. A derived predicate is a predicate which is defined solely by derivation rules. A derived (base) literal is one whose predicate is derived (base). The set of derivation rules is also known as the intensional database or program. In respect to relational databases, the set of all ground instances of a predicate \( p \) with arity \( n \) can be thought of as a relation, the \( p \) relation, in which each ground instance corresponds to a tuple of the \( p \) relation and each column corresponds to an argument of \( p \). The extension of an atom \( p(t_1, t_2, \ldots, t_n) \) is the subset of the \( p \) relation such that each tuple corresponds to a ground instance of \( p(t_1, t_2, \ldots, t_n) \). We will often name a set of columns of the extension of \( p \) by the set of variables occupying the corresponding argument positions in \( p \), so that a projection on columns \( \chi \) of the extension of \( p \) is denoted by \( \pi_\chi p \) or \( p[\chi] \). For simplicity of exposition, and without loss of generality, we have chosen to rewrite databases so that

1. a predicate is either base or derived, but not both, so that a database of the form
   
   \[
   \begin{align*}
   a(1,1) \\
   a(X,Y) &\leftarrow b(X,Y) \\
   a(X,Y) &\leftarrow b(X,Z), a(Z,Y)
   \end{align*}
   \]

   is rewritten as
   
   \[
   \begin{align*}
   c(1,1) \\
   a(X,Y) &\leftarrow c(X,Y) \\
   a(X,Y) &\leftarrow b(X,Y) \\
   a(X,Y) &\leftarrow b(X,Z), a(Z,Y);
   \end{align*}
   \]

2. constants in derived literals are moved into new equality atoms, so that a rule of the form
   
   \[
   a(1,Y) \leftarrow b(X,2), c(X,Y), \neg s(3, Y)
   \]

   is rewritten as
   
   \[
   a(T1,Y) \leftarrow T1 = 1, T2 = 2, b(X, T2), c(X,Y), T3 = 3, \neg s(T3, Y).
   \]

A deductive database \( \mathbf{D} \) (or simply database) is a finite set of rules consisting of a program \( \mathbf{P} \) and a set of facts \( \mathbf{F} \). The term database includes the possibility of negative body literals. In some instances we differentiate between two types of databases: when the rules in the database do not contain any negative body literals, we refer to it as a positive database and to the program as a positive program; when this is not the case, we refer to a normal database and a normal program.

The following are borrowed (and slightly modified due to the absence of function symbols) from [29]. An interpretation for \( \mathbf{D} \) consists of the following:

1. a nonempty set \( \mathcal{D} \), called the domain of interpretation, and an assignment of constants in \( \mathbf{D} \) to elements in \( \mathcal{D} \); and

2. for each \( n \)-ary predicate in \( \mathbf{D} \), a mapping from \( \mathcal{D}^n \) into \{true, false\}.

Each rule or query can be written in terms of a first-order formula. An interpretation for a formula \( S \) is called a model for \( S \) if \( S \) evaluates to true under the
interpretation. The *Herbrand universe* corresponding to a database \( D \) is the set of constants and predicate names appearing in \( D \). The *Herbrand base* corresponding to a database \( D \) is the set of all ground atoms which can be formed by using predicates from \( D \) with ground terms from the Herbrand universe as arguments. A *Herbrand interpretation* for \( D \) is one where

1. the domain of the interpretation is the Herbrand universe;
2. constants in \( D \) are assigned to “themselves” in the Herbrand universe.

For a program \( P \), we construct a dependency graph \( \mathcal{G} \) representing a *relationship* between the predicates. This is a directed graph where there is a node for each predicate and an arc from node \( q \) to node \( p \) if there is a body literal whose predicate is \( q \) in a rule whose head predicate is \( p \). When this literal is negative, the arc is a *negative arc* and is marked with a prime; otherwise it is a *positive arc* and is unmarked. A predicate \( p \) *depends on* a predicate \( q \) if there is a path of length greater than or equal to one from \( q \) to \( p \). We denote the relation \( p \) *depends on* \( q \) by \( p \leadsto q \), where \( \leadsto \) is the transitive closure of the *refers to* relation. If any arc in the path from \( q \) to \( p \) is negative, then we may also denote the dependency by \( p \leftarrow q \) (\( p \leftarrow q \Rightarrow p \leftarrow q \)). A predicate \( p \) is *recursive* if \( p \leftarrow q \). Two predicates \( p \) and \( q \) are *mutually recursive* if \( p \leftarrow q \) and \( q \leftarrow p \). Note that diagrams of \( \mathcal{G} \) in this paper do not show base predicates, since we are concerned with potential cycles, and base predicates cannot be part of a cycle.

**Example 1.** The dependency graph corresponding to the program

\[
\begin{align*}
p(X,Y) & \leftarrow q(X,Y) \\
p(X,Y) & \leftarrow q(X,Z), p(Z,Y) \\
h(X,Y) & \leftarrow \neg p(X,Z), p(Z,Y)
\end{align*}
\]

is shown in Figure 1.

A *strongly connected component* of a graph \( \mathcal{G} \) is a subgraph \( \mathcal{G}_s \) of \( \mathcal{G} \) such that there is a path \( (m_i, m_j) \) of nonzero length between each pair of nodes \( m_i, m_j \) in \( \mathcal{G}_s \). Each maximal subgraph \( \mathcal{G}_s \) is called an *MSCC* (maximal strongly connected component).

2. **BOTTOM-UP EVALUATION**

In this section we briefly review the standard bottom-up techniques as they apply to both positive and normal deductive databases.
2.1. Positive Databases

Bottom-up methods for answering a query to a positive database are similar to the application of the \( T \) operator [47]. Essentially, they are refinements of the following scheme. Initialize a set of ground atoms \( S \) with the database facts. Repeatedly apply the rules to \( S \) until no new atoms are generated. A rule is applied to \( S \) by adding the head of a ground instance of the rule to \( S \) whenever all the body literals of this instance are in \( S \). The answers to the query are those instances of the query in the final set \( S \). The scheme terminates in the absence of function symbols because the number of ground atoms is finite. It is well known that each application step in the bottom-up method can be efficiently implemented using techniques from relational databases. An immediate refinement is to consider only potentially relevant rules.

**Definition 1.** Define \( \text{pred}(p) \) to be a function which returns the predicate name corresponding to the literal \( p \).

**Definition 2.** For a database \( D \), a query \( \leftarrow q \), and a rule \( R \in D \),

\[ p_0 \leftarrow p_1, \ldots, p_m, \quad m \geq 0, \]

we say \( R \) is potentially relevant to \( q \) if \( \text{pred}(q) \) is the same as \( \text{pred}(p_0) \) or if \( \text{pred}(q) \leftarrow \text{pred}(p_0) \).

In general, bottom-up computations may generate the same tuple many times. The computation can be further refined with the differential method suggested in [10, 7, 5]. Using this approach, the tuples generated during each iteration are constructed by exploiting the new tuples generated during the previous iteration. In [6], we have generalized the differential method for nonlinear recursive rules.

2.2. Normal Databases; Stratified Databases

The class of stratified databases [14] was introduced to make the model theory manageable when negative body literals are included in the database, by disallowing certain combinations of recursion and negation that cause dependencies of the form \( p \leftarrow p \).

**Definition 3.** A negative cycle is a cycle in \( \mathcal{G} \) where at least one arc in the cycle is negative.

**Definition 4.** A database \( D \) is stratified if and only if there does not exist a negative cycle in \( \mathcal{G} \) of \( D \).

The algorithm for determining whether a database is stratified is linear in time and space with respect to the size of the database [28].

**Definition 5.** A stratification of \( D \) is a partitioning of the rules of \( D \) into the sets \( D_0, \ldots, D_n \) such that the following conditions hold for \( i = 0, \ldots, n \):

1. if a predicate \( p \) occurs in \( D_i \) as a positive body literal, then its definition is contained in \( \bigcup_{j \leq i} D_j \).
(2) if a predicate \( p \) occurs in \( D_i \) as a negative body literal, then its definition is contained in \( \bigcup_{j<i} D_j \).

Each set \( D_i \) is called a stratum, and each \( i \) is called a level.

**Proposition 1.** \( D \) has a stratification if and only if it is stratified.

**Proof.** See [1]. \( \square \)

The usual \( T \) operator [47] is no longer viable for stratified databases, since \( T \) is not monotonic and a unique least fixed point may not exist (even in the absence of recursion). A model-based semantics for stratified programs is presented in [1], based on iterative fixed-point operators that compute a particular minimal model. A similar independent treatment also appeared in [48].

Consider any stratification \( D_0, \ldots, D_n \) of a database \( D \). Let \( M \) denote a set of ground atoms. Define \( T_i \), \( i = 0, \ldots, n \), to be the operator on \( M \) as follows. For every \( M \) and every ground atom \( p \), \( p \in T_i(M) \) if and only if \( p \in M \) or for some rule \( p_0 \leftarrow p_1, \ldots, p_m \) in \( D_i \), there is a substitution \( \theta \) of constants for variables such that \( p = p_0\theta \) and for each body literal \( p_j \), if \( p_j \) is positive then it is in \( M \), otherwise it is not in \( M \).

The sets \( M_0, \ldots, M_n \) of ground atoms are defined by the equations

\[
M_0 = \emptyset,
M_i = T_i^j(M_{i-1}) \quad (i = 1, \ldots, n),
\]

where \( T_i^j \) is \( T_i \) applied \( j \) times until the fixpoint is reached for some value of \( j \geq 0 \).

The intended model of \( D \), which we take as the semantics of \( D \), is \( M_n \). This model, which we write as \( M(D) \), does not depend on the choice of stratification of \( D \) [1] and is identical to the perfect model [36].

**Example 2.** As an illustration of how we might go about implementing this semantics using a bottom-up computation, consider the following program and query in which \( b, c \) and \( d \) are base predicates.

(1) \( p(X,Y) \leftarrow b(X,Y) \)
(2) \( p(X,Y) \leftarrow s(X,Z), d(X,Z), p(Z,Y) \)
(3) \( s(X,Y) \leftarrow c(X,Y) \)
(4) \( s(X,Y) \leftarrow c(X,Z), s(Z,Y) \)
(5) \( p(X,Y). \)

Owing to the stratification of the database, the set of tuples satisfying \( s \) is evaluated before the set of tuples satisfying \( p \). So first we use rules (3) and (4), as we would for a positive database, to derive the \( s \) tuples. This has the effect of making \( s \) a pseudo base relation when we come to derive the \( p \) tuples using rule (2). The predicate \( p \) is recursive, and so at each iteration potentially new \( p \) facts are generated. For the first iteration, we add all the \( b \) facts to the \( p \) tuples. After this we repeatedly apply rule (2) until no more new \( p \) tuples are found. We apply rule (2) by first joining \( p \) and \( d \) (based on \( Z \)), obtaining some \( \langle X,Y,Z \rangle \) tuples. We then remove any \( \langle X,Y,Z \rangle \) tuple for which a corresponding \( \langle X,Z \rangle \) tuple of \( s \) has been found. Next we project on \( X \) and \( Y \) from the remaining \( \langle X,Y,Z \rangle \) tuples and add the projected \( \langle X,Y \rangle \) tuples to \( p \).
2.3. Domain-Independent Databases

Example 3. Consider the following database and query:

\[
\begin{align*}
\text{human}(peter) \\
\text{animal}(X) & \leftarrow \neg \text{human}(X) \\
& \leftarrow \text{animal}(X)
\end{align*}
\]

The only constant is peter. If we apply the \( T_i \) operator, above, \( M(D) \) is \( \{\text{human}(peter)\} \). There are no ground \( animal \) atoms in \( M(D) \), and if we consider the answer to a query as being those ground atoms in \( M(D) \) which are unifiable with the query, the answer to \( \leftarrow \text{animal}(X) \) is false.

There is an implicit assumption governing the computation of \( M(D) \) using the \( T_i \) operator, namely, that the language is determined by the constants in the database at the time the query is asked. In the database context, there are problems with this assumption. If we introduce a new fact to the database, such as \( \text{flower}(rose) \), \( M(D) \) will now be

\[
\{\text{human}(peter), \text{animal}(rose), \text{flower}(rose)\}
\]

and the value of \( X \) which satisfies \( \leftarrow \text{animal}(X) \) is rose. As pointed out in [45], the fact that the answers to a query can "change" in this fashion is undesirable. Databases which do not exhibit this behavior are called domain-independent databases.

2.4. Allowedness

Domain independence is a model-theoretic property. A simple sufficient syntactic characterization of domain-independent databases is given by the notion of allowedness. Similar ideas expressed with respect to first-order queries to a relational database are range-separable [16], range-restricted [34], and safe formulas [46].

Several different definitions of allowedness have appeared in the literature. Clark's original definition [15] is equivalent to Shepherdson's subsequent covering axiom [43], and is the most common. We borrow the wording in [27].

Definition 6. A rule is Clark-allowed if every variable appears in at least one positive body literal.

Definition 7. A database is Clark-allowed if every rule in the database is Clark-allowed.

2.5. Implementation Considerations

Practical complications occur when allowedness is not present. The bottom-up computation of a rule in a Clark-allowed program can be divided into two processes. Consider a rule

\[
P \leftarrow p_1, \ldots, p_m, \neg s.
\]

First the set of \( p_i \) tuples, \( 1 \leq i \leq m \), satisfying the conjunction \( p_1, \ldots, p_m \) is generated. Let this be denoted by the relation \( P \). Next, those tuples satisfying the
positive literal \( s \), which we denote by the \( S \) relation, are removed using a
set-difference operation. By the allowedness property we know that the columns of
\( S \) are a subset of the columns of \( P \). The difference operation is then
\[ P := P \setminus (P \bowtie S). \]
If a rule were not allowed, for example,
\[ p(X,Y) \leftarrow r(X), \neg s(Y), \]
we could not use set difference, since the operation would be inapplicable.

Another problem with disallowed rules is that under the standard quantification
discussed in [29] the negative literal corresponds to an existential query \( \exists Y \neg s(Y) \),
and there are various possibilities for deciding what the domain of \( Y \) values ought
to be. Even if we were to choose a domain, for example the Herbrand universe of
the database, we must subtract the \( s \) tuples from this domain. Typically, this result
would compute a much larger relation than \( s \). In practice rules of this form are not
desirable and are proscribed. Top-down computations are said to “flounder” for
such rules. With an allowed database, however, we are not forced to represent
such a large relation representing negative information, because the domain is
determined by the positive body literals and is smaller.

3. THE ADORNED DATABASE

The first step in the magic-set transformation is to produce the adorned database
[46]. The adorned database is a formal way of depicting information flow between
literals in the database. This is done by annotating predicates with a character
string. The adornment of the literal \( p(t_1, t_2, \ldots, t_n) \) is a string made up of the
letters \( b \) and \( f \) which is attached to, and becomes part of the predicate name \( p \).
The string is defined by the following mapping:

1. \( b \) represents the word bound, and \( f \) represents the word free.

2. During a computation, each argument \( t_i, 1 \leq i \leq n \), of the literal
   \( p(t_1, t_2, \ldots, t_n) \) is expected to be bound or free, depending on the information
   flow. If \( t_i \) is expected to be bound (free), it acquires a \( b \) (\( f \)) annotation,
   and so the length of the adornment string is \( n \).

An adornment \( bff \) of \( p(X,Y,Z) \) is denoted by \( p^{bff}(X,Y,Z) \). The predicate
\( p^{bff} \) is called an adorned predicate. An advantage of the adornment formalism is
that it permits us to describe a query form [21]. A query form is nothing more than
an adorned query predicate. It is a generic representation of the set of queries
which can be asked using the query predicate.

**Definition 8.** The adorned query atom corresponding to a query \( q(t_1, t_2, \ldots, t_n) \),
is written as \( q^a(t_1, t_2, \ldots, t_n) \), where the adornment \( a \), which is a string \( a_1 \ldots a_n \)
of \( b \) and \( f \), is defined as follows: if \( t_i \) is a constant, then set \( a_i \) to \( b \); otherwise
set \( a_i \) to \( f \).

As described in [12], adornments are generated with reference to a specific
sideways information-passing strategy. The adorned database can therefore be
thought of as mirroring the binding information specified by the SIPS.
3.1. Sideways Information-Passing Strategy: Motivation

Example 4. Consider the familiar ancestor predicate where ancestor(X, Y) is true if Y is an ancestor of X, and where parent is a base predicate, such that parent(X, Y) is true if Y is a parent of X:

\begin{align*}
(1) & \quad \text{ancestor}(X, Y) \leftarrow \text{parent}(X, Y) \\
(2) & \quad \text{ancestor}(X, Y) \leftarrow \text{parent}(X, Z), \text{ancestor}(Z, Y).
\end{align*}

The query \( \leftarrow \text{ancestor}(\text{talya}, Y) \) retrieves all the ancestors of talya. A standard bottom-up computation proceeds by computing the entire ancestor relation and then selecting the tuples whose first argument is talya. Note that we cannot solve this problem by syntactically pushing the constant talya into the rules by replacing all occurrences of X in the program with talya, because the set of answers to the query is no longer identical to that of the original program. Thus, the bottom-up method—either in its pure form or using the differential approach—is likely to be inefficient when the query atom or body literals contain ground arguments, since many irrelevant tuples are generated. The reader is referred to [9] or [19] for a comparison of different methods to solve this problem.

Contrast this with the way that a PROLOG or breadth-first computation (see [44] for example) dynamically pushes constants from the query into the rules, and from one body literal to the next, after each literal has been evaluated. Continuing with our example, the query atom \( \text{ancestor}(\text{talya}, Y) \) is unified with the head of the first rule. The variable X is bound to talya and is used in the body of (1) to retrieve Talya’s parents. Similarly for rule (2), except that here the resultant Z value is available to \( \text{ancestor}(Z, Y) \) when it is evaluated. The values for Z can then be said to be passed sideways from parent to ancestor.

A formal method for expressing the SIPS that are desirable during query evaluation is described in [12]. The information (set of bound values) that is passed to a literal \( p \) permits \( p \) to be solved with respect to those bindings. SIPS formally describe what information is passed by one literal, or a conjunction of literals, to another literal. It is important to stress that SIPS do not say how this information is passed. Indeed, there may be more than one way to pass the information for given SIPS. For example, SIPS do not specify whether the information is passed on a tuple-at-a-time basis, or as a set of tuples. SIPS subsume the dataflow graphs used in the analysis of backtracking schemes and elsewhere. They are a generalization of the sideways propagation graphs described in [26].

Returning to the example above, the information passing we described was induced because the query was \( \leftarrow \text{ancestor}(\text{talya}, Y) \). A different strategy is applicable if the query is \( \leftarrow \text{ancestor}(X, \text{talya}) \), since here different information will be passed from one literal to another. SIPS are associated with a rule according to the query form. Different query forms, such as \( \text{ancestor}^{bf}(X, Y) \) and \( \text{ancestor}^{fo}(X, Y) \), usually have different SIPS for the same set of defining rules. The choice of one SIPS over another is guided by factors such as the current and expected size of the different relations and the indexing mechanism employed. For the purposes of this paper, we assume, throughout, that this choice has been made based on such considerations.
In order to differentiate between the SIPS corresponding to different query forms, we explicitly adorn the head predicate in the SIPS according to the adornment derived from the query.

3.2. Formal Definition

Definition 9. Two literals $p$ and $s$ are connected if

1. $p$ and $s$ share a common argument;
2. inductively, if $p$ is connected to $r$ and $r$ is connected to $s$, then $p$ is connected to $s$.

Definition 10. Let $B(R)$ be the set of body literals for a rule $R$, and let $p^a$ be the adorned head literal of $R$ for some adornment $a$. A SIPS for $R$ is a labeled bipartite graph $\mathcal{G}(V_1, V_2)$, where $V_1$ is the set of subsets of $B(R) \cup \{p^a\}$, where $V_2 = B(R)$, and where the following two conditions hold:

1. Each arc is of the form $\mathcal{N} \rightarrow \chi \ s$, where $\mathcal{N} \in V_1$, $s \in V_2$. The label $\chi$ stands for a nonempty set of variables which satisfies the following conditions:
   i. each variable in $\chi$ appears in
      a. $s$,
      b. either a bound argument position of $p^a \in \mathcal{N}$ or a positive body literal in $\mathcal{N}$ (or both);
   ii. each literal in $\mathcal{N}$ is connected to $s$.
2. There exists a total order of $B(R) \cup \{p^a\}$ in which:
   i. $p^a$ precedes all members of $B(R)$;
   ii. any literal which does not appear in the graph follows every literal that appears in the graph; and
   iii. for each arc $\mathcal{N} \rightarrow \chi \ s$, if $s' \in \mathcal{N}$, then $s'$ precedes $s$.

3.3. Examples of SIPS

We now present some examples of SIPS which illustrate their use.

Example 5. Consider the program

\[
\begin{align*}
  a(X, Y) &\leftarrow b(X, Y, Z) \\
a(X, Y) &\leftarrow b(X, Z, W), a(Z, T), c(T, W), a(T, Y).
\end{align*}
\]

Let the query be $\leftarrow a(1, Y)$. This corresponds to the adorned head predicate $a_{bf}$ and adorned query $\leftarrow a_{bf}(1, Y)$. An arc for the first rule, corresponding to this adornment, might be

\[
\{a_{bf}(X, Y)\} \rightarrow_{(X)} b(X, Y, Z).
\]
Consider two SIPS for the second rule:

\[
\begin{align*}
\{a^{bf}(X, Y)\} &\rightarrow_{\{X\}} b(X, Z, W) \\
\{a^{bf}(X, Y), b(X, Z, W)\} &\rightarrow_{\{Z\}} a(Z, T) \\
\{a^{bf}(X, Y), b(X, Z, W), a(Z, T)\} &\rightarrow_{\{T, W\}} c(T, W) \\
\{a^{bf}(X, Y), b(X, Z, W), a(Z, T), c(T, W)\} &\rightarrow_{\{T\}} a(T, Y), \\
\end{align*}
\]

(I)

\[
\begin{align*}
\{a^{bf}(X, Y)\} &\rightarrow_{\{X\}} b(X, Z, W) \\
\{b(X, Z, W)\} &\rightarrow_{\{Z\}} a(Z, T) \\
\{a(Z, T)\} &\rightarrow_{\{T\}} c(T, W) \\
\{c(T, W)\} &\rightarrow_{\{T\}} a(T, Y). \\
\end{align*}
\]

(II)

There is a subtle difference between the two SIPS. In (I) the last three arcs specify that each body literal in the head of the SIPS receives some information based on the evaluation of a conjunction of body literals to its left. However, with (II), although \(a(Z, T)\) passes ground values for \(T\) to \(c(T, W)\), the ground values for \(W\) that are generated need not be the same set of ground values for \(W\) computed by evaluating \(b(X, Z, W)\) from the previous arc. It is only when all the tuples in the conjunction of body literals are joined at the end that the compatible values of \(W\) from \(c\) and \(b\) are reconciled (unified).

Example 6. As an example of SIPS which include a negative literal, consider the rule

\[
a(X, Y) \rightarrow \neg s(X, Y), r(Y), t(X).
\]

Let the query be \(\leftarrow a(1, Y)\). One possible strategy is

\[
\begin{align*}
\{a^{bf}(X, Y)\} &\rightarrow_{\{X\}} \neg s(X, Y) \\
\{a^{bf}(X, Y), \neg s(X, Y), r(Y)\} &\rightarrow_{\{X\}} t(X).
\end{align*}
\]

In the first arc, a bound value for \(X\) is passed to \(\neg s(X, Y)\). That is, the corresponding positive query \(s(X, Y)\) is evaluated with \(X\) bound. The ground value for \(X\) is then passed to \(t\) provided that there exists a \(\langle Y \rangle\) tuple in \(r\) such that \(\langle X, Y \rangle\) is not a tuple of \(s\).

An alternative strategy might be

\[
\begin{align*}
\{a^{bf}(X, Y), r(Y)\} &\rightarrow_{\{X, Y\}} \neg s(X, Y) \\
\{a^{bf}(X, Y), r(Y), \neg s(X, Y)\} &\rightarrow_{\{X\}} t(X).
\end{align*}
\]

Here, when the answers to \(\neg s(X, Y)\) are inferred, the corresponding positive query \(s(X, Y)\) is asked, with both \(X\) and \(Y\) being bound. Only the \(X\) values for which a \(Y\) value for \(r\) such that \(\langle X, Y \rangle\) is not a tuple of \(s\) are passed to \(t\).

It is possible to design SIPS which are not easily implementable. For the program of Example 6, the SIPS are

\[
\{ \neg s(X, Y) \} \rightarrow_{\{X\}} t(X)
\]

is not sensible. A negative literal removes or restricts the flow of information (derived from positive literals). A negative literal does not generate information as
Algorithm 1. Constructing the adorned database based on SIPS

function adorn \((q^a, P, S)\)
\[
N := q^a \quad P^a := S^a := A := O
\]
while \(N \neq \emptyset\) do
\[
\text{remove an adorned predicate } p^a \text{ from } N
\]
\[
A := A \cup \{p^a\}
\]

let \(R^p\) be a copy of the set of rules defining \(p\) in \(P\)
for each rule \(R \in R^p\) with head \(p(\bar{e})\) do
\[
\text{let } S(R) \text{ be a copy of the SIPS associated with } R \text{ for the adorned head } p^a
\]
replace \(p(\bar{e})\) by \(p(\bar{e})^a\) in \(R\)
for each derived body literal \(u\) in \(R\) do
\[
\hat{a} := \text{adornment}(u, S(R))
\]
let \(t\) be the predicate name of \(u\)
replace \(t\) in both \(R\) and \(S(R)\) by \(t^a\)
if \(t^a \notin A\) then \(N := N \cup \{t^a\}\)
\[
\text{od}
\]
\[
P^a := P^a \cup (R)
\]
\[
S^a := S^a \cup \{S(R)\}
\]
\[
\text{od}
\]
\[
\text{return } (q^a, P^a, S^a)
\]

this arc implies. We will define the concept of legal SIPS in Section 5.3. For the present it suffices, as defined in [11] and presented in Definition 10, that each variable in the label of an arc appears in a positive literal of the tail \(N\).

3.4. Adornment Algorithm

Given a query to a database, the problem can now be recast as that of answering the query relevantly with reference to sideways information-passing strategies under the bottom-up paradigm. The method that we have developed to implement SIPS is based upon a magic-set transformation of the adorned database. We proceed first by adorning the query, as described earlier, and continue by adorning body literals. For a database \(D\), set of SIPS \(S\), and adorned query \(q^a(\bar{t})\), the adorned program \(P^a\) and associated SIPS \(S^a\) are constructed by the \textit{adorn} function in Algorithm 1 using two data structures: the set \(N\) of adorned predicates \(p^a\) such that the rules defining \(p\) have not yet been adorned with \(a\) and the set \(A\) of adorned predicates \(p^a\) such that the rules defining \(p\) have been adorned with \(a\).

Definition 11. Given \(n\) SIPS arcs, \(n \geq 1\), leading to a literal \(p\):
\[
N_1 \rightarrow x_1 \ p
\]
\[
N_2 \rightarrow x_2 \ p
\]
\[
\vdots
\]
\[
N_n \rightarrow x_n \ p.
\]
The normalized arc for \(p\) is
\[
\{N_1 \cup N_2 \cup \cdots \cup N_n\} \rightarrow \bigcup_{1 \leq i \leq n} x_i \ p.
\]
We simplify the presentation of *adorn* by assuming that all SIPS have been normalized. The following function is used in *adorn*:

\[ \text{adomment}(u, s) \] takes a body literal \( u \) and associated SIPS \( s \), and returns a string \( a_1, \ldots, a_n \) of \( b's \) and \( f's \) (adornment) according to the following rule. Let \( N \rightarrow u \) be an arc in \( s \); \( \chi \) is effectively \( \emptyset \) if no such arc exists. Let the arity of \( u \) be \( n \). For each argument \( x_i \) in the \( i \)th argument position of \( u \), if \( x_i \in \chi \) then set \( a_i \) to \( b \); otherwise set \( a_i \) to \( f \).

The subset of rules in \( D \) that are adorned by the algorithm are those corresponding to the program \( P \). The adorned database \( D'^u \) is made up of the adorned program \( P'^u \) output by *adorn* and the facts \( F \). We make two remarks about the algorithm.

The algorithm does not adorn base body literals, although it could be adapted to do so. For the purposes of this paper, the focus is on a magic transformation to reduce the number of irrelevant tuples. Since base relations are fully constructed, the retrieval mechanism (which could make use of adornments to base literals) returns only relevant tuples in any case.

Recall from Section 1.1 that constants from derived predicates are transferred to new equality atoms. Without loss of generality, we assume that these are the only equality atoms in the program and that there are no incoming arcs to them. Equality atoms must therefore contain at least one constant, e.g. \( X = 1 \), and so we treat them as base literals in the adornment algorithm.

### 3.5. Equivalence

A number of different types of equivalence have been discussed in the literature [31, 42, 13, 22]. Sagiv uses two terms: *equivalence* and *uniform equivalence* (these were also independently introduced by Maher).

**Definition 12.** Program \( P_1 \) contains \( P_2 \), written as \( P_2 \subseteq P_1 \), if for every finite set of ground atoms \( F \) corresponding to base predicates, \( M(P_2 \cup F) \subseteq M(P_1 \cup F) \).

**Definition 13.** Two programs \( P_1 \) and \( P_2 \) are equivalent, written as \( P_1 = P_2 \), if \( P_2 \subseteq P_1 \) and \( P_1 \subseteq P_2 \).

**Definition 14.** Program \( P_1 \) uniformly contains \( P_2 \), written as \( P_2 \subseteq^u P_1 \), if for every finite set of ground atoms \( F \) corresponding to base or derived predicates, \( M(P_2 \cup F) \subseteq M(P_1 \cup F) \).

**Definition 15.** Two programs \( P_1 \) and \( P_2 \) are uniformly equivalent, written as \( P_1 =^u P_2 \), if \( P_2 \subseteq^u P_1 \) and \( P_1 \subseteq^u P_2 \).

We introduce a form of equivalence called *equivalence with respect to a query* and show that the transformations described in this paper preserve this equivalence. For the remainder of the paper we consider only equivalence with respect to a query, because we have defined our databases so that \( F \) consists only of facts
corresponding to base predicates. However, our transformations equally preserve uniform equivalence with respect to a query.

Definition 16. Let $P_1$ be the program associated with $D_1$, and let $P_2$ be the program associated with $D_2$. Let $F$ be an arbitrary finite set of (base) facts such that $D_1 = P_1 \cup F$ and $D_2 = P_2 \cup F$. Let $\bar{t}$ denote an arbitrary term (which may contain constants and variables), and let $\bar{c}$ denote an arbitrary ground term. Let $\leftarrow q_1(\bar{t})$ be a query on $D_1$, and $\leftarrow q_2(\bar{c})$ be a query on $D_2$ (the query predicates are not necessarily distinct). Two programs $P_1$ and $P_2$ are equivalent with respect to $q_1$ and $q_2$, written as $P_1 \equiv _{q_1} P_2$, when $q_1(\bar{c}) \in M(D_1)$ if and only if $q_2(\bar{c}) \in M(D_2)$.

Proposition 2. Let $D$ be any stratified database with associated program $P$, and let $\leftarrow q(\bar{t})$ be a query. Let $D^a$ be the corresponding adorned database and $P^a$ the adorned program, and let $\leftarrow q^a(\bar{t})$ be the adorned query. Then $P \equiv _{q^a} P^a$.

Proof. For a proof see [11].

We now show the adorned programs for both SIPS (I) and (II), corresponding to the program of Example 5 and adorned query $\leftarrow a^{bf}(1, Y)$.

Example 5 (Continued).

\[
a^{bf}(X, Y) \leftarrow b^{bf}(X, Y, Z)
\]

$\leftarrow b^{bf}(X, Z, W), a^{bf}(Z, T), c^{bf}(T, W), a^{bf}(T, Y)$.

Adorned program for SIPS (I)

\[
a^{bf}(X, Y) \leftarrow b^{bf}(X, Y, Z)
\]

$\leftarrow b^{bf}(X, Z, W), a^{bf}(Z, T), c^{bf}(T, W), a^{bf}(T, Y)$.

Adorned program for SIPS (II)

Example 6 (Continued). For the SIPS of Example 6, the respective adorned rules are

\[
a^{bf}(X, Y) \leftarrow \neg s^{bf}(X, Y), r(Y), t^b(X)
\]

and

\[
a^{bf}(X, Y) \leftarrow s^{bf}(X, Y), t^b(X), r(Y).
\]

4. A NEW MAGIC-SET ALGORITHM

In this section we consider the magic-set transformation of positive databases only. Magic-set algorithms are program transformations that take an initial adorned database and query and return a modified database which gives the same answers for a particular query as the initial database. Using the bottom-up method, the transformed database generates fewer irrelevant tuples than the initial database. There have been several magic-set algorithms reported in the literature [8, 12, 41]. The independently developed Alexander method [39] is essentially the method in [12] with the improvements described in [41].

A common trait amongst the more recent algorithms is that, based on the adornment of the head and body literals, new positive literals are introduced into
the body of rules, and new rules are added to the database which define these literals. The new literals are called \textit{magic literals} and are related to the existing literals of the database as follows. For a positive adorned predicate \( p^a \) with \( m \) bound argument positions where \( m > 0 \) (i.e., there are \( m b's \) in the adornment of \( p^a \)), define the \textit{magic predicate} of \( p^a \) to be the predicate whose name is the predicate name of \( p^a \) prefixed with "magic-" and whose arity is \( m \). The new rules defining the magic predicates are called \textit{magic rules}.

A note on the name: the term \textit{magic set} refers to the set of tuples constructed for a magic predicate during bottom-up computation of the transformed database. Before presenting the magic-set algorithm, we refine the concept of irrelevant computation by first defining the \textit{set of relevant ground rules} \( g(D) \).

\textbf{Definition 17.} For any rule \( R \) in \( D \), a (Herbrand) \textit{ground instance} of \( R \), denoted by \( R_g \), is constructed by applying some substitution \( \theta \), of constants from the Herbrand universe, to all the variables in \( R \).

\textbf{Definition 18.} The \textit{set of ground rules} of a database \( D \), denoted by \( g(D) \), is the subset of the ground instances of rules in \( D \) such that for each \( R_g \in g(D) \), the positive body literals of \( R_g \) are in \( M(D) \).

\textbf{Definition 19.} The \textit{set of relevant ground rules} for a query \( \leftarrow q \) to \( D \) is the subset \( g^q(D) \) of \( g(D) \) constructed as follows:

1. Initialize \( g^q(D) \) with each rule in \( g(D) \) whose head literal is an instance of \( q \).
2. Recursively, \( g^q(D) \) contains each rule in \( g(D) \) whose head literal appeared as a positive body literal in a rule in \( g^q(D) \).

\textbf{Definition 20.} The \textit{relevant tuples} for a query \( \leftarrow q \) to \( D \) is the set of tuples corresponding to the set of heads of rules in \( g^q(D) \).

\textbf{Definition 21.} The computation of answers to a query \( \leftarrow q \) to a database \( D \) is \textit{irrelevant} if it uses a ground rule \( R_g \) such that \( R_g \notin g^q(D) \).

Irrelevant tuples are usually generated, since the system does not know in advance whether or not a tuple is relevant. The common idea underlying magic-set techniques is that generating magic tuples is worthwhile because it saves generating many irrelevant tuples.

The magic-set algorithm we propose is simpler and more efficient than existing algorithms. Its simplicity comes about because we view the arcs of SIPS associated with a rule as magic rules written in reverse.

The following definitions are used in the magic-set algorithm:

- \textit{bodyLit} (\( N \)) denotes the conjunction of body literals of a rule \( R^a \) in \( N \), where \( N \) is the tail of an arc \( N \rightarrow x \ p \) in the SIPS for \( R^a \).
- \textit{Magic} (\( p^a(t) \)) returns a literal \( \text{magic}_p^a(t_a) \), where \( t_a \) is the vector of arguments which are bound in the adornment \( a \) of \( p \).

\textbf{Proposition 3.} For a positive adorned database \( D^a \) and adorned query \( \leftarrow q^a \), let \( P^m \) be the program transformed by the magic-set transformation of Algorithm 2, so that \( D^m = P^m \cup \text{Magic}(q^a) \cup F \). Then \( P^a \equiv q^a P^m \).
ALGORITHM 2. Magic-set transformation for positive databases

function magic_set \( q^a(\overline{v}), P^a, S^a \) 
\[ P^m := \emptyset \]
for each rule \( R^a \) in \( P^a \) of the form \( h \leftarrow B \) do
add the rule \( h \leftarrow Magic(h), B \) to \( P^m \)
for each arc \( A \rightarrow v \), \( p \) in the SIPS associated with \( R^a \) do
if \( h \) is in \( A \) then
add the rule \( Magic(p) \leftarrow Magic(h), bodyLit(A) \) to \( P^m \)
else
add the rule \( Magic(p) \leftarrow bodyLit(A) \) to \( P^m \)
fi
od
od
return \( (Magic(q^a(\overline{v})), P^m) \)

PROOF. See Section A.1 in the Appendix. \( \square \)

4.1. An Example

For the remainder of the paper we omit explicit SIPS for simplicity of exposition, and assume default SIPS where the tail of the arc for each body literal \( p \) includes all literals to the left of \( p \) in the rule (including the head). The predicates \( g, b, h, \) and \( c \) are base predicates. Corresponding "real world" examples can be found in [3].

Example 7. Consider the adorned query \( \leftarrow e^b(randy) \) to the program
\[ a(X) \leftarrow g(X) \]
\[ a(X) \leftarrow b(X, Y), a(Y) \]
\[ e(X) \leftarrow c(X), a(X), h(X, Y), c(Y), a(Y). \]
The corresponding adorned program is
\[ a^b(X) \leftarrow g(X) \]
\[ a^b(X) \leftarrow b(X, Y), a^b(Y) \]
\[ e^b(X) \leftarrow c(X), a^b(X), h(X, Y), c(Y), a^b(Y). \]
The magic rules and modified rules are
\[ a^b(X) \leftarrow magic_a^b(X), g(X) \]
\[ a^b(X) \leftarrow magic_a^b(X), b(X, Y), a^b(Y) \] (*)
\[ magic_a^b(Y) \leftarrow magic_a^b(X), b(X, Y) \] (†)
\[ magic_a^b(X) \leftarrow magic_e^b(X), c(X) \]
\[ magic_a^b(Y) \leftarrow magic_e^b(X), c(X), a^b(X), h(X, Y), c(Y) \]
\[ e^b(X) \leftarrow magic_e^b(X), c(X), a^b(X), h(X, Y), c(Y), a^b(Y) \]
\[ magic_e^b(randy). \]

REMARK. An extension of the GMS algorithm to the case where there are multiple arcs to a predicate such as
\[ \{c^b_l(X, Y), a^b(X)\} \rightarrow_p (X, Y) \]
and
\[ \{ c^{bf}(X,Y), b^{b}(Y) \} \rightarrow_{\{Y\}} p(X,Y) \]
is given in [12]. Essentially, a magic rule is created in the usual way corresponding
to each arc. The head is not named a magic predicate. Instead it is given a label as in
\[
\begin{align*}
label_1 p(X) & \leftarrow c^{bf}(X,Y), magic_{a^{b}}(X), a^{b}(X) \\
label_2 p(Y) & \leftarrow c^{bf}(X,Y), magic_{b^{b}}(Y), b^{b}(Y).
\end{align*}
\]
The magic rule for \( p \) is composed of the two new predicates as
\[
\begin{align*}
magic_{p^{bb}}(X,Y) & \leftarrow label_1 p(X), label_2 p(Y).
\end{align*}
\]
Because we normalize SIPS, we can handle the situation of multiple arcs leading
to a given literal in a more natural way. We do not require special label predicates.
By a process of unfolding it is easy to see that the solution using special labels is
equivalent to normalizing the arcs before the adornment stage and applying our
magic-set algorithm. The rule is simply
\[
\begin{align*}
magic_{p^{bb}}(X,Y) & \leftarrow c^{bf}(X,Y), a^{b}(X), b^{b}(Y).
\end{align*}
\]
Normalization of arcs is preferable, since it introduces no extra predicates and the
magic-set algorithm is further simplified. Any optimization of the join inherent in
the body of the magic rule is dealt with using the techniques developed for
relational databases.

4.2. Supplementary Magic Sets

It should be apparent that some further optimization of the magic-transformed
program is possible using common subexpression elimination. For example, the
body of the rule marked (†) in Example 7 is reevaluated in the body of the rule
marked (*). An algorithm to remove such redundancies which uses supplementary
magic sets was introduced in [41] and generalized in [12]. We note that, although
we do not present the supplementary magic version of our examples, in practice all
programs should be further optimized in this way. A simple approach to imple-
menting a magic- and supplementary-magic-set type optimization without employ-
ing a program transformation technique is described in [35].

5. MAGIC SETS AND ALLOWEDNESS

In this section we examine the applicability of magic-set algorithms to databases
whose rules include negative body literals. Suppose we apply the magic-set algo-
rithm to a stratified and (Clark) allowed database. One could expect that the
transformed rules would also be both stratified and allowed. However, as we will
show, neither property is necessarily preserved. We confine ourselves to the
question of allowedness in this section and defer the discussion of stratification to
the next section.

An obvious way to apply the magic-set transformation to a normal program is to
make a slight change to the algorithm to admit arcs which lead to negative literals.
The revised transformation for normal databases is presented in Algorithm 3. Unless otherwise specified, when we refer to a magic-set algorithm we mean this algorithm, although the discussion is equally applicable to the GMS algorithm in [11].

5.1. Preserving Allowedness
Recall that the adornments derived from SIPS are a statement of what information is passed during a computation. In this paper, we have focused on bottom-up computation as the mechanism specifying how that information is passed. In the context of a bottom-up computation, magic-set transformations facilitate certain sideways information-passing strategies. After performing the magic transformation, a bottom-up computation is performed, based on the family of $T$ operators of the transformed program.

We wish to preserve the syntactic property of allowedness because it permits a simplified approach to handling negative literals as described in Section 2.4 and guarantees domain independence. Several different definitions of allowedness have appeared in the literature. Clark's original definition (in Definition 6 above) is more restrictive than the one we propose, because his definition is independent of a given query. The definition in [30] is an attempt to deal with the query. It considers whether the rule representing the query is also allowed. However, it does not take into account the adornments generated by that query.

5.2. A New Definition of Allowedness
We introduce a new definition which is less restrictive than previous ones in that it can take into account specific information from the adorned database as derived from the query form.

Definition 22. An adorned rule is allowed if every variable appears in a positive body literal or a bound argument position of the adornment of the head.
Intuitively, the domains of free arguments in the head and arguments of negative body literals are determined by the positive body literals of the body and the bound arguments of the head.

Definition 23. An adorned database is allowed if every adorned rule in the database is allowed.

A given database may be allowed for one query and choice of SIPS, but disallowed for another. We will prove in Section 7.5 that a stratified and allowed (using our definition) program is domain-independent. At this stage we note, informally, that for a top-down computation, whenever a rule is invoked, the bound positions of the head of a rule are replaced by constants. Since all other variables in the rule appear in positive literals, the invocation conforms to Clark-allowedness and retains the property of domain independence.

Proposition 4. Determining whether an adorned database is allowed can be done while the adorned database is constructed, in linear time with respect to the size of the adorned database.

The proof is straightforward.

For the remainder of this paper we assume the new definition of allowedness unless explicitly stated otherwise. To see that our definition of allowedness is less restrictive than the query-independent ones mentioned above, consider the following Clark-disallowed rule:

\[ \text{likes}(X,Y) \leftarrow \text{expensive}(Y), \neg \text{possesses}(X,Y). \]

The predicate \text{possesses} is derived, whereas \text{expensive} is a base predicate. Let the query be \( \leftarrow \text{likes}^{bf}(\text{john},Y) \), and the arc (from the SIPS) be

\[ \{\text{likes}^{bf}(X,Y), \text{expensive}(Y)\} \Rightarrow_{\{(X,Y)\}} \neg \text{possesses}(X,Y). \]

The adorned rule

\[ \text{likes}^{bf}(X,Y) \leftarrow \text{expensive}(Y), \neg \text{possesses}^{bb}(X,Y) \]

is allowed using our definition, since \( X \) in \text{possesses} is bound in the head predicate \text{likes} and \( Y \) is bound by the base predicate \text{expensive}.

5.2. Allowed SIPS

Essentially, the magic-set algorithm does two things: it creates new magic rules, and it introduces new body literals into the original rules to form modified rules. For the modified rules, since the heads remain unchanged and no new negative body literals are introduced, allowedness is preserved. However, the new rules, which include fragments of the bodies of the original rules, are not always allowed. Given a query \( \leftarrow h^b(1,Y) \), consider the following (Clark) allowed rule:

\[ h(X) \leftarrow \neg p(X,Y), a(X), b(Y) \]
with associated SIPS arcs
\[
\{ h^b(X) \} \rightarrow_{(X)} \neg p(X,Y),
\]
\[
\{ h^b(X), \neg p(X,Y) \} \rightarrow_{(X)} a(X),
\]
where \( h \) is a base predicate, and \( a \) and \( p \) are derived predicates. The magic rule for \( a \) (derived from this rule) is not allowed:
\[
magic\_a^b(X) \leftarrow magic\_h^b(X), \neg p(X,Y).
\]

Practical complications occur when allowedness is not preserved. Let us assume the standard implicit quantification discussed in [29]. A bottom-up computation of \( \leftarrow magic\_a^b(X) \) would have to solve the existential subquery \( \leftarrow \exists Y \neg p(X,Y) \). Several possibilities exist with regard to choosing a domain for \( Y \). In practice rules of this form are not desirable and are proscribed. The possibility of such rules being generated can be removed if we restrict ourselves to allowed SIPS. The definition is similar to that of SIPS. The changes are in the refining of the original condition (1X(i)) in Definition 10 to the new condition (1X(iii)).

**Definition 24.** Let \( B(R) \) be the set of body literals for a rule \( R \), and let \( p^a \) be the adorned head literal for some adornment \( a \). An allowed SIPS for a rule \( R \) is a labeled bipartite graph \( \mathcal{G}(V_1, V_2) \), where \( V_1 \) is the set of subsets of \( B(R) \cup \{ p^a \} \) and \( V_2 = B(R) \), and which satisfies the following two conditions:

1. Each arc is of the form \( N \rightarrow \chi \ q \), where \( N \in V_1, q \in V_2 \). The label \( \chi \) stands for a nonempty set of variables which satisfies the following conditions:
   1. each variable in \( \chi \) appears in a member of \( N \) and in \( q \);
   2. each literal in \( N \) is connected to \( q \);
   3. each variable appearing in \( N \) appears in a positive literal in \( N \), or in a bound argument position of \( p^a \) in \( N \).

2. There exists a total order of \( B(R) \cup p^a \) in which:
   1. \( p^a \) precedes all members of \( B(R) \);
   2. any literal which isn’t in the graph follows every literal that is in the graph; and
   3. for every arc \( N \rightarrow \chi \ q \), if the literal \( q' \in N \), then \( q' \) precedes \( q \).

**Proposition 5.** If we apply the magic-set transformation of Algorithm 3 to an adorned allowed database \( D^a \) and allowed SIPS \( S^a \), then it returns an adorned allowed database, \( D^m \).
Magic(p) appears in a body literal of \( R^m \). We now show that each variable in a body literal of \( R^m \) appears in a positive body literal. The body consists of the literals in \( \mathcal{N} \) and possibly the magic literal Magic(h). If \( \mathcal{N} \) contains a negative literal, then by condition (1)(iii) in the definition each variable in the negative literal either has a b adornment in \( h \), or is in a positive literal in \( \mathcal{N} \). If the variables are in positive literals in \( \mathcal{N} \), then \( R^m \) is clearly allowed, since the body of \( R^m \) contains \( \mathcal{N} \). If the variables are in a bound argument position of \( h \), then they will be in a positive body literal of \( R^m \) because the body of \( R^m \) will include Magic(h).

A similar concept to allowed SIPS was introduced in [11]. The original condition as described in [12] for positive databases is each variable of \( \chi \) appears in \( \mathcal{N} \).

This was revised for normal databases in [11] to each variable of \( \chi \) appears in ... a positive member of \( \mathcal{N} \) [our emphasis].

As demonstrated by the following example, this condition is not sufficient to ensure a Clark-allowed transformed rule.

**Example 8.** Consider the following (Clark) allowed rule:

\[
h(X) \leftarrow \neg p(X,Y), a(X), b(Y)
\]

and corresponding SIPS (which is disallowed by our definition)

\[
\{h^b(X), \neg p(X,Y)\} \rightarrow_{\{X\}} a(X),
\]

where \( b \) is a base predicate, and \( a \) and \( p \) are derived predicates.

The magic rule for \( a \) is not allowed:

\[
\text{magic}_{a^b}(X) \leftarrow \text{magic}_{h}(X), b(X), \neg p^{bf}(X,Y).
\]

For the remainder of this paper we consider only allowed SIPS. There is an interesting relationship between our definition of allowedness in terms of adornments and Clark's definition. When a database is transformed using the magic-set algorithm, the original rules are modified to contain new (positive) magic literals whose arguments correspond to the bound argument positions in the adornment of the head. As a consequence, in determining whether a modified rule is allowed, it is sufficient to check that every variable appears in at least one positive body literal. This is precisely the definition of allowedness in the sense of Clark. Similarly, since we use allowed SIPS, the magic rules are also allowed in this sense.

**Proposition 6.** The magic-set transformation of Algorithm 3 applied to an adorned allowed database \( D^a \) and allowed SIPS \( S^a \) and query \( \leftarrow q^a \) results in a Clark-allowed database, \( D^m \).

**Proof.** By the proof of Proposition 5, the magic rules and the facts are Clark-allowed. Consider the modified rules. Let \( R \) be a derivation rule and \( R' \) be the corresponding modified rule. By our definition of allowedness a variable in the head of \( R \) which is not also in a positive body literal must be in a bound argument...
position of the adornment of the head. By construction $R'$ is identical to $R$ except
that it has an additional positive magic body literal whose arguments include all
the variables in a bound argument position of the head. Therefore each variable in
$R'$ appears in a positive body literal, and $R'$ is Clark-allowed. ☐

6. STRATIFICATION AND MAGIC RULES FOR POSITIVE LITERALS

The magic-set transformation does not always preserve stratification. In this
section we examine the cause of unstratification. We will do this by initially
focusing on magic rules constructed for positive body literals. That is, we will ignore
for the moment the magic rules constructed for negative literals. Of course, the
transformation is not complete—our initial aim though is to discover causes.

Definition 25. Consider the line marked (*) in Algorithm 3. If the algorithm
constructs a magic rule corresponding to an arc from the SIPS of the form
$\mathcal{N} \rightarrow _x p$, then we say that a magic rule is constructed for a positive body literal. If
the algorithm constructs a magic rule corresponding to an arc from the SIPS of
the form $\mathcal{N} \rightarrow _x \neg p$, then we say that a magic rule is constructed for a negative
body literal.

Example 9. Consider the following program:

\begin{align*}
e(X) & \leftarrow c(X), a(X), h(X,Y), c(Y), a(Y) \\
a(X) & \leftarrow g(X) \\
a(X) & \leftarrow b(X,Y), a(Y).
\end{align*}

For the query \( \leftarrow e^b(randy) \) the magic rules and modified rules are

\begin{align*}
e^b(X) & \leftarrow \text{magic}_e^b(X), c(X), a^b(X), h(X,Y), c(Y), a^b(Y) \\
\text{magic}_e^b(randy) \\
a^b(X) & \leftarrow \text{magic}_a^b(X), g(X) \\
a^b(X) & \leftarrow \text{magic}_a^b(X), b(X,Y), a^b(Y) \\
\text{magic}_a^b(Y) & \leftarrow \text{magic}_a^b(X), b(X,Y) \\
\text{magic}_a^b(X) & \leftarrow \text{magic}_e^b(X), c(X) \\
\text{magic}_a^b(Y) & \leftarrow \text{magic}_e^b(X), c(X), a^b(X), h(X,Y), c(Y).\end{align*}

The dependency graph is shown in Figure 2. There are three magic rules corre-
spending to the predicate $a^b$. The first is due to the body literal $a^b$ in the
definition of $a^b$ itself. The second and third rules, however, are derived from the
body predicates $a^b$ in the rule defining $e^b$. Although there are three magic rules
for $a^b$, there is one magic set constructed for $a^b$, the tuples satisfying \text{magic\_a^b}.
We could easily number each occurrence of $a^b$ and treat each body literal and its
subqueries independently. The implicit assumption of the magic-set transforma-
tion, however, is that it is better to compute one larger magic set for $a^b$ than three
smaller ones corresponding to each instance of $a^b$ in the body.
Example 10. We now modify the previous program by adding the rule
\[ f(X) \leftarrow d(Y), \neg a(X), h(X, Y), a(Y) \]
and give the query \( \leftarrow f^b(petra) \). The transformed rules are
\[
\begin{align*}
  f^b(X) & \leftarrow \text{magic}^b_f(X), d(X), \neg a^b(X), h(X, Y), a^b(Y) \\
  \text{magic}^b_f(petra) & \\
  a^b(X) & \leftarrow \text{magic}^b_a(X), g(X) \\
  a^b(X) & \leftarrow \text{magic}^b_a(X), b(X, Y), a^b(Y) \\
  \text{magic}^b_a(Y) & \leftarrow \text{magic}^b_a(X), b(X, Y) \\
  \text{magic}^b_a(Y) & \leftarrow \text{magic}^b_f(X), d(X), \neg a^b(X), h(X, Y) \\
  \text{magic}^b_a(X) & \leftarrow \text{magic}^b_f(X), d(X). 
\end{align*}
\]

The third rule (in bold), defining \( \text{magic}^b_a \), which is derived from the positive literal \( a(Y) \) in the rule defining \( f \), introduces the negative cycle
\[ a^b \leftarrow \text{magic}^b_a \leftarrow a^b \]
into the dependency graph of Figure 3.

FIGURE 2. Dependency graph of transformed program of Example 9.

FIGURE 3. Dependency graph of transformed program of Example 10.
ALGORITHM 4. Labeling algorithm

/* the \( L_i, I_i \) and \( C_i \) are global variables */
define function label\( (L_1, \ldots, L_n) \)

- initialize \( I_1, \ldots, I_n \) to \( \emptyset \)
- initialize the \( C_1, \ldots, C_n \) to 0

neglabel( )
poslabel( )

\( P' := \bigcup_{i=1}^{n} (L_i \cup I_i) \)

\( S' := \bigcup_{i=1}^{n} (S_i \cup s_i) \)

return\( (P', S') \)

Let us examine the source of this unstratification by considering a top-down computation of the rules. There are two contexts in which the predicate \( a \) appears in the body of the rule defining \( f \). In the first, it is part of a negative literal, whereas in the second, it is a part of a positive literal. For the query \( \leftarrow f^b(X) \), a top-down computation will evaluate the subquery \( a^b(X) \) using an \( X \) which satisfies \( d(X) \). If the subquery fails, then the negative literal \( \neg a^b(X) \) succeeds. This value for \( X \) satisfies \( f \) provided that \( h(X, Y) \) is proven and a value for \( Y \) satisfies the subquery \( a^b(Y) \). The important point is that the two queries to \( a \)—the negative subquery and the positive subquery—are independent of each other. That is, at the time that \( a^b(Y) \) is asked, \( \neg a^b(X) \) has already been satisfied. When we construct magic rules for \( a \), however, this separation of contexts is lost. The magic rules for a positive literal and a negative literal are treated as one, when they should be separated and used in accordance with the context in which they are required. The mixing of contexts underlies the unstratification problem and is addressed by Algorithm 4.

6.1. The Labeling Algorithm

The key concept behind the labeling algorithm is to distinguish the context for constructing magic sets. The approach is to explicitly label \( p \) when it appears as a negative body literal in a rule \( R \). We only label the predicates of those positive literals that appear in the defining rules for the new negatively labeled predicates. We employ a maximal stratification in the algorithm in order to simplify the analysis in Section A.1 of the Appendix.

Definition 26. For a database \( D \), with program \( P \), a stratification \( L_0, \ldots, L_n \) is maximal if for every stratum \( L_i, 1 \leq i \leq n \), either

1. \( L_i \) contains exactly the rules defining a predicate \( p \) if \( p \) is not recursive, or
2. \( L_i \) contains exactly the rules defining \( p \) and (any) other predicates in the same MSCC as \( p \) if \( p \) is recursive; and
3. \( L_0 \) contains all the base facts.

The input to the labeling algorithm consists of \( P^a \) and the corresponding set of SIPS \( S^a \). The program \( P^a \) is first arranged into a maximal stratification so that \( P^a = \bigcup_{i=1}^{n} L_i \) and the associated SIPS are \( S^a = \bigcup_{i=1}^{n} S_i \), where each set of SIPS \( S_i \) is associated with the rules in \( L_i \). Collectively, the rules and SIPS for a stratum \( L_i \) are denoted by \( L_i^a \). During execution of the labeling algorithm, each stratum \( L_i \) has an associated set of newly constructed rules \( I_i \) and their SIPS \( s_i \). Collectively, the
ALGORITHM 5. Negative labeling procedure

procedure neglabel ()
for i := 1 to n do
    for each q ∈ negBodyLits(i) do
        replace each q in negative body literals of L_i by n_q
    od
    copy L_i to L'_i
    replace each predicate p defined in stratum i with n_p throughout L'_i
od
end

new rules and SIPS created during the algorithm are denoted by L'_i, i = 1, ..., n.
The output of the algorithm is the labeled program P', where P' = \bigcup_{i=1}^{n}(L'_i ∪ L_i),
and associated SIPS S'. A resultant labeled database D' is then just P' ∪ F.

There are two stages in Algorithm 4. In the first stage, we negatively label those
predicates which appear as negative body literals. We create new rules defining the
newly labeled predicates. In the second stage we positively label these new rules
and create further new rules for the new positively labeled predicates.

Definition 27. A predicate n_q in a labeled program P' is negatively labeled if it
was formed by replacing a negative occurrence of q in the unlabeled adorned
program P^u.

Definition 28. A predicate p_k in a labeled program P', where k is an integer, is
positively labeled if it was formed by replacing an occurrence of p in the
unlabeled adorned program P^u.

The first stage of the algorithm calls neglabel in Algorithm 5. The procedure
performs two actions. First it examines each stratum and negatively labels the
negative body literals. Secondly, for each predicate p, it creates a negatively
labeled version of the rules defining p (n_p). Note that this is done at compile
time, regardless of whether p actually appears negatively in the program, so that a
subsequent user query which may involve \neg p has a compiled version of its rules at
hand. In [3] we called this facility query-independent compilation. We make use of
the following set:

negBodyLits(i) is the set of predicates that appear as negative body literals in the
stratum L_i.

After neglabel has executed we call the poslabel procedure (Algorithm 6). The
purpose of neglabel was to create new predicates corresponding to negative literals
\neg n_s so as to separate these new predicates from the positive literal s. However,
in order to completely separate the rules defining n_s from the rules defining s, we
label derived positive body literals p such that n_s ← p. We associate a single
counter C_i with each stratum, L_i. So if p is defined in stratum C_i, it is replaced by
ALGORITHM 6. Positive labeling procedure

\begin{verbatim}
procedure poslabel()
    for i := 1 to n - 1 do
        for each unlabeled positive derived literal \( p \) appearing in \( L_i \) do
            replace \( p \) by \( p_{\alpha} \), where \( \text{defined}_n(p, m) \) and where \( \alpha = C_m + 1 \)
        od
    for j := i - 1 downto 1 such that \( \text{depends}(i, j) \) do
        make a copy of \( L_j \) called \( L_j' \)
        for each unlabeled positive derived literal \( q \) appearing in \( t_j' \) do
            replace \( q \) by \( q_{\alpha} \), where \( \text{defined}_n(q, k) \) and where \( \alpha = C_k + 1 \)
        od
        add \( L_j' \) to \( L_j \)
        \( C_j := C_j + 1 \)
    od
end
\end{verbatim}

\( p_{\alpha} \), where \( \alpha \) is derived from \( C_i \). The following new functions are used by \textit{poslabel}:

- \( \text{depends}(i, j) \) returns \textit{true} if there exists a path in \( \mathcal{G} \) from a predicate defined in \( L_j \) to a predicate defined in \( L_i \); otherwise it returns \textit{false}.
- \( \text{defined}_n(p, i) \) sets \( i \) to be the stratum number in which the predicate \( p \) is defined.

6.2. Sample Trace

\textit{Example 11}. It may not be obvious why negative labeling on its own is not sufficient. In the following example we indicate why, and at the same time provide a sample trace of \textit{label}. We omit variables and SIPS for simplicity. Consider the program

\[
\begin{align*}
L_3 & \quad p \leftarrow \neg s, a, r \\
L_2 & \quad s \leftarrow r \\
L_1 & \quad r \leftarrow b.
\end{align*}
\]

The \( L_i \) form the maximal stratification. After the magic transformation the rules

\[
\begin{align*}
p & \leftarrow \text{magic}_p, \neg s, a, r \\
s & \leftarrow \text{magic}_s, r \\
r & \leftarrow \text{magic}_r, b \\
\text{magic}_r & \leftarrow \text{magic}_p, \neg s, a \\
\text{magic}_r & \leftarrow \text{magic}_s.
\end{align*}
\]

The dependency graph with the negative cycle \( s \leftarrow r \leftarrow \text{magic}_r \leftarrow s \) is shown in Figure 4. The negative literal in the cycle is \( \neg s \).
First, we execute `negluble`. The resultant strata are:

- $L_3$: $p \leftarrow \neg n_s, a, r$
- $L_2$: $s \leftarrow r$
- $L_1$: $r \leftarrow b$
- $I_3$: $n_p \leftarrow \neg n_s, a, r$
- $I_2$: $n_s \leftarrow r$
- $I_1$: $n_r \leftarrow b$.

Before we perform `poslab`, note that the dependency graph of the relevant magic transformed program of the resultant strata

- $p \leftarrow magic_p, \neg n_s, a, r$
- $r \leftarrow magic_r, b$
- $magic_r \leftarrow magic_p, \neg n_s, a$
- $magic_r \leftarrow magic_n_s$
- $n_s \leftarrow magic_n_s, r$

is shown in Figure 5 and still contains the cycle $n_s \leftarrow r \leftarrow magic_r \leftarrow n_s$. We only display relevant derived predicates in the dependency graph. Also note that $p$ and $r$ are not included in the graph.

**FIGURE 4.** Dependency graph for the magic program of Example 11.

**FIGURE 5.** Dependency graph for the negatively labeled magic program of Example 11.
$r$ are negatively labeled even though they are not needed (for this program and query combination).

At this point we execute \textit{poslabel}. That is, we label the positive body literals in the rules defining (newly) negatively labeled predicates. After \textit{poslabel} the final strata are

$$L_3 \quad p \leftarrow \neg n_s, a, r$$
$$L_2 \quad s \leftarrow r$$
$$L_1 \quad r \leftarrow b$$
$$L_3 \quad n_p \leftarrow \neg n_s, a, r$$
$$L_2 \quad n_s \leftarrow r_1$$
$$L_1 \quad n_r \leftarrow b \quad r_1 \leftarrow b.$$

The \textit{relevant} program is

$$L_3 \quad p \leftarrow \neg n_s, a, r$$
$$L_2 \quad n_s \leftarrow r_1$$
$$L_1 \quad r \leftarrow b \quad r_1 \leftarrow b.$$

The magic-transformed program is

$$p \leftarrow \text{magic}_p, \neg n_s, a, r$$
$$n_s \leftarrow \text{magic}_n_s, r_1$$
$$r \leftarrow \text{magic}_r, b$$
$$r_1 \leftarrow \text{magic}_r_1, b$$
$$\text{magic}_r \leftarrow \text{magic}_p, \neg n_s, a$$
$$\text{magic}_r_1 \leftarrow \text{magic}_n_s.$$

The dependency graph of the relevant predicates is shown in Figure 6 and the cycle is removed.

\textbf{FIGURE 6.} Dependency graph for the labeled magic program of Example 11.
Proposition 7. Let $\mathbf{D}'$ be the resultant database after applying the labeling algorithm to a stratified database $\mathbf{D}$. The database $\mathbf{D}'$ is stratified, and for any query $\mathbf{P} \equiv_q \mathbf{P}'$.

The proof of this proposition is similar to that of Proposition 2 and can be found in [2].

Proposition 8. For a stratified database $\mathbf{D}$, if $\mathbf{D}'$ is the resultant database after applying the labeling algorithm to $\mathbf{D}$, and $\mathbf{D}''$ is the resultant database after applying the magic-set transformation of Algorithm 3 (constructing magic rules only for positive literals) to $\mathbf{D}'$, then $\mathbf{D}''$ is stratified.

PROOF. See Section A.2 in the Appendix. 

Note that although $\mathbf{P}''$ is stratified, it is not equivalent to $\mathbf{P}''$, since we have not constructed the magic rule for the negative literal.

6.3. Magic Rules for Negative Literals

By implementing context separation, the labeling algorithm ensures that stratification is preserved when the magic rules for positive body literals are constructed. In order to see that stratification is not necessarily preserved when we include the magic sets constructed for negative literals. Consider a modification of Example 9.

Example 12. Given the query $\mathbf{P} \equiv i(sandy)$ to the program

\begin{align*}
  s(X) & \leftarrow g(X) \\
  s(X) & \leftarrow b(X,Y), s(Y) \\
  i(X) & \leftarrow k(X) \\
  i(X) & \leftarrow \neg s(X), j(X,Y), i(Y).
\end{align*}

The relevant transformed rules after labeling and magic transformation are

\begin{align*}
  i^b(X) & \leftarrow \text{magic}_i^b(X), k(X) \\
  i^b(X) & \leftarrow \text{magic}_i^b(X), \neg n_s^b(X), j(X,Y), i^b(Y) \\
  \text{magic}_i^b(sandy) & \\
  \text{magic}_i^b(Y) & \leftarrow \text{magic}_i^b(X), \neg n_s^b(X), j(X,Y) \\
  n_s^b(X) & \leftarrow \text{magic}_n_s^b(X), g(X) \\
  n_s^b(X) & \leftarrow \text{magic}_n_s^b(X), b(X,Y), n_s^b(Y) \\
  \text{magic}_n_s^b(Y) & \leftarrow \text{magic}_n_s^b(X), b(X,Y) \\
  \text{magic}_n_s^b(X) & \leftarrow \text{magic}_i^b(X).
\end{align*}

The negative dependency

\begin{align*}
  n_s^b & \leftarrow \text{magic}_n_s^b \leftarrow \text{magic}_i^b \leftarrow n_s^b
\end{align*}

in Figure 7 incorporates the dependencies derived from the magic rule constructed for the negative literal $\neg n_s(X)^b$ (in bold) above.

We conjecture that no prior transformation of the original database without a new approach to bottom-up execution is sufficient to preserve the stratification of
the magic transformed database. We now describe our solution to this problem—a structured bottom-up computation—which builds on the theoretical properties provided by the labeling concept (without actually performing the labeling), and takes into account magic rules constructed for negative body literals, at the same time preserving the efficiency of magic sets and the perfect model semantics.

7. STRUCTURED BOTTOM-UP METHOD

7.1. Motivation

One way of answering a query to a magic-transformed set of rules is to restrict the transformation to positive literals. That is, we remove arcs leading to a negative literal before applying the adornment algorithm, so that negative literals are evaluated without any magic-set transformation on their defining rules. As we have proven, the labeled and magic-transformed program is stratified and allowed, and hence semantically well defined. It is not efficient, because in answering a negative literal $\neg n_s$ we compute the entire extension for $n_s$ (no bound argument positions in the adornment). On the other hand, if we do attempt to use the arcs leading to $\neg n_s$ and construct the magic rules for $\neg n_s$, the resultant program is not necessarily stratified and the semantics unclear. Normally, the stratification property defines an order of execution. Presented with unstratification, we do not know which order to apply the rules in, or whether one order is “better” than another. This dilemma requires a solution.

The purpose of a magic literal in the body of a derivation rule is simply to restrict the search for tuples satisfying the head. Given a negative literal $\neg n_s$ and some bindings consistent with its adornment, we seek to compute the extension of $n_s$. Recall that the extension of $n_s$ is a relation which corresponds to the set of all ground instances of $n_s$. Next we compare the extension of $n_s$ with the extensions of the remaining body literals and perform the negation operation using set difference.
The search space used in the construction of \( n_s \) can be restricted by using magic sets, provided that we are sure not to lose any relevant tuples. Specifically, we can utilize the magic set so that instead of computing the full extension of \( n_s \), only that part of the extension which is determined by the given bindings is computed. Thus, the magic literals and their defining rules are seen to have an auxiliary role with respect to the rules of the untransformed program. We can determine an order of use for both the magic and transformed rules by considering the computation of the untransformed program, and how bound values are propagated within it.

Whenever we reach a rule containing the body literal \( \neg n_s \), the bindings for the variables labeling the arc \( \mathcal{N} \rightarrow x \) \( \neg n_s \) are available by evaluating the conjunction \( \mathcal{N} \). Relating this to the magic-transformed program, in theory, we ought to be able to invoke a query \( \leftarrow n_s \) on the magic-transformed rules defining \( n_s \) (minus the magic rule constructed for \( \neg n_s \)) by constructing initial magic facts from the bindings of the set of variables \( x \) labeling the arc. A bottom-up computation of the program segment of rules defining \( n_s \) could then take place. The ground instance of \( n_s \) computed by the program segment can then be combined with the data already accumulated for the rule in which \( \neg n_s \) was a body literal. The \( n_s \) tuples would then be subtracted and the original computation could continue.

The set of rules defining \( n_s \) can be thought of as disconnected from the rest of the program by virtue of the fact that the connecting magic rule constructed from \( \neg n_s \)—which would normally provide initial ground instances of magic \( n_s \)—is not included. The justification for assuring that a computation of rules defining \( n_s \) is disconnected or separate from those rules defining (positive) \( s \) is the labeling algorithm. Predicates that are used in the context of answering \( n_s \) do not influence the computation of those which are used to answer \( s \), when \( s \) appears as a positive body literal.

In practical terms, it would be better if a solution did not actually have to perform labeling. The solution we now propose effectively performs the same separation of contexts as labeling but without having to "physically" label the predicates before the magic-set algorithm is applied. We achieve this in the same way that a common compiler implements a function call.

There is a strong connection between the function-call paradigm and the labeling paradigm. For each negative literal, labeling effectively creates a new separate program defining the literal. With a function call, we can organize the (no longer labeled) program so that in the context of answering \( \neg s \), a function will exclusively use a subset of the rules in the program. This subset of rules corresponds precisely to the rules that would have been created using a labeling approach. The solution defines a correct, modified bottom-up computation which permits ground instances of \( s \) to be initialized in the context of answering \( \neg s \), so that the program is no longer incomplete.

### 7.2. Function-Call Paradigm

Considering a query \( \leftarrow p \) to the program (ignoring variables)

\[
\begin{align*}
p & \leftarrow \neg s, s \\
s & \leftarrow s, a \\
s & \leftarrow a.
\end{align*}
\]
We define a main program as performing a bottom-up computation of the set of transformed rules defining \( p \). We assume that magic rules for a negative literal \( \neg s \) have not been constructed. For each such negative literal \( \neg s \) which appears in the body of the rules in the main program, we construct a program segment consisting of a copy of the modified rules that are needed to define \( s \). Next, we add some rules defining magic_s. Not all rules defining magic_s are added. Recall that a property of the labeling algorithm is that magic rules for \( s \) are constructed exclusively from those rules which have \( s \) as a body literal and whose head is in the same MSCC as \( s \). The transformed main program would include all the rules required to define \( p - p + magic_p, s \), \( p + magic_p \), \( s \), \( s + magic_s \), \( s \), \( a \), \( s + magic_s, a \), \( magic_s + magic_s, a \), \( magic_s + magic_s \).

Note that the last rule isn't necessarily a tautology, since it could abbreviate a rule such as

\[ magic_s^{bb}(X,Y) \leftarrow magic_s^{bb}(Y,X). \]

The program segment used to define \( s \) in the context of answering \( \neg s \) consists of

\[ s \leftarrow magic_s, a \]
\[ s \leftarrow magic_s, s, a \]
\[ magic_s \leftarrow magic_s. \]

These correspond exactly to the rules defining \( n.s \) had we used a labeling approach. Note that we did not include the magic rule in bold, above, because it was derived from a rule whose head \( p \) is not in the same MSCC as \( s \). Since \( s \) appears both positively and negatively in the main program, we simply duplicate the rules defining \( s \) for the program segment, as above, in the knowledge that these rules will be used exclusively when answering \( s \) in the context of a negative literal. We remark that this duplication is a meager overhead, since at worst, although the original number of derivation rules in the program may be doubled, this is typically many orders of magnitude less than the number of rules in the database. If this increase proved prohibitive, a simple scheme involving program sharing could always be devised in a practical implementation.

Informally, the computation proceeds as follows. The main program and the program segment used to define \( s \) in the context of answering \( \neg s \) are defined as above. When the rules in the program segment are called to answer \( \neg s \), the execution of the main calling program is temporarily halted and an initialization of "input variables" occurs. In our case the input is the set of \( magic_s \) facts determined by the set of variables \( \chi \) in the arc leading to \( \neg s \) in the calling program. Execution then proceeds in the program segment using its private copy of the rules defining \( s \). The answers to the positive query \( s \) for the given input bindings are found, and control returns to the calling program, after which a set difference of \( s \) tuples will be performed. By implementing the eval function
presented in Algorithm 7 in this way, a dynamic separation of contexts, equivalent to the static labeling-based approach, is achieved.

7.3. The DAG-Structured Database

We now describe the reorganization of a database so that it reflects the "program segment" approach we just described. First we refine the definition of relevant rules given in Definition 2. Recall that this definition was given with respect to positive databases. Now we must consider stratified databases. We said that a predicate \( p \) depends on a predicate \( q \) if there is a path of length greater or equal to one from \( q \) to \( p \) in the dependency graph of a database. If no arc in a path from \( q \) to \( p \) is negative, then we denote the dependency by \( p \leftarrow q \) \((p \leftarrow q \Rightarrow p \leftarrow q)\).

Definition 29. Given a predicate \( q \) and database \( D \), the set of positively relevant rules of \( q \), denoted by \( prules(q) \), is made up of all rules \( R \in D \) of the form

\[
p_0 \leftarrow p_1, \ldots, p_m, \quad m \geq 0,
\]

for which \( q = \text{pred}(p_0) \), or \( q \leftarrow \text{pred}(p_0) \).

Formally, the database is organized as a directed acyclic graph (DAG) as follows.

Definition 30. For an adorned, stratified, and allowed database \( D^a \) and adorned query \( \leftarrow q(l) \), let \( D'' \) be the adorned and magic-set-transformed database corresponding to \( D^a \) without including the magic rules for negative literals. Let \( \mathcal{D}(D'', q) \) denote the negation-induced DAG defined as follows. Each node \( p \), where \( p \) is a predicate name, contains \( prules(p) \). The root node, corresponding to the query, contains \( prules(q) \). Recursively, there is an adjacent node \( s \) containing \( prules(s) \) and path from \( q \) to \( s \) for each distinct derived predicate \( s \) appearing negatively in \( prules(q) \).

The important thing to note about the DAG is that it is acyclic. To see this, note that because we have removed the magic rules for negative literals, the rules for a node \( s \), where \( s \) appears in a negative body literal are precisely those which correspond to the set of rules which have defined \( n \ s \ had we used labeling. By Proposition 8 these rules are stratified, only here we achieve the same effect as labeling by creating an adjacent node for each negative derived literal. All that remains is to be sure that the execution of rules in a mode \( q \) and its adjacent node \( s \) preserves the desired context separation implied by labeling. This is done dynamically by the \( \text{eval} \) function, as we will demonstrate.

Example 13. The negation-induced graph for Example 12 has two nodes: \( i^b \) and \( s^b \). The node for \( prules(i^b) \) is

\[
i^b(X) \leftarrow \text{magic}_i^b(X), k(X)
i^b(X) \leftarrow \text{magic}_i^b(X), \neg s^b(X), j(X, Y), i^b(Y)
\]

\[
\text{magic}_i^b(\text{sandy})
\]

\[
\text{magic}_i^b(Y) \leftarrow \text{magic}_i^b(X), \neg s^b(X), j(X, Y).
\]
The (negatively induced) node for $\text{prules}(s^b)$ is
\[
\begin{align*}
    s^b(X) &\leftarrow \text{magic}_s(s^b(X), g(X)) \\
    s^b(X) &\leftarrow \text{magic}_s(s^b(X), b(X,Y), s^b(Y)) \\
    \text{magic}_s(s^b(Y)) &\leftarrow \text{magic}_s(s^b(X), b(X,Y)).
\end{align*}
\]

REMARK. Although technically base facts are elements of $\text{prules}(n)$, in practice they would not be represented explicitly in the node. This explains the absence of facts for the base predicates $k$ and $j$ in the root node $i^b$, above, and $g$ and $b$ for the negatively induced adjacent node $s^b$.

7.4. The eval Function

Definition 31. Consider the conjunction of body literals
\[
    \mathcal{B} = p_1, \ldots, p_k, \neg s_1, \ldots, \neg s_j
\]
for a Clark-allowed rule $R$. We define the extension of $\mathcal{B}$, $\text{ext}(\mathcal{B})$, as a relation whose columns are named by the variables in $p_1, \ldots, p_k$. Each variable in an $s_i$ appears in a positive literal. This is assured by the Clark-allowedness property of the magic-transformed rules. The extension is the result of the following algebraic operations. Perform a join of the positive literals. The join conditions are obtained in the usual way by considering common variables and constants. We denote the result of this join by $\mathcal{P}$. Next, for each negative literal $s_i$, $1 \leq i \leq j$, remove the $s_j$ tuples from $\mathcal{P}$. That is, for each $i$, $1 \leq i \leq j$, we iteratively execute
\[
    \mathcal{P} := \mathcal{P} \setminus (\mathcal{P} \bowtie \text{ext}(s_i)).
\]

The resultant $\mathcal{P}$ is the extension of $\mathcal{B}$.

Let us denote a projection $\pi_{\mathcal{P}}$ over a relation $\mathcal{P}$ as projecting over all the columns of $\mathcal{P}$. Recall that we name a set of columns of the extension of $\mathcal{P}$ by the set of variables occupying the corresponding argument positions in $\mathcal{P}$, so that a projection on columns $\chi$ of the extension of $\mathcal{P}$ is denoted by $\pi_{\chi} \mathcal{P}$ or $\mathcal{P}[\chi]$. Given two relations $\mathcal{P}$ and $\mathcal{J}$, and a set of common attributes $\chi$, rather than performing
\[
    \pi_{\mathcal{P}}(\mathcal{P} \bowtie \mathcal{J}),
\]
the well-known semijoin optimization performs
\[
    \mathcal{P} \bowtie_{\chi} \mathcal{J}.
\]

Let us now denote the relation which is to be subtracted from $\mathcal{P}$ by $\mathcal{J}$. As described by Equation (1), in computing the extension of $\mathcal{B}$ we execute the set difference
\[
    \mathcal{P} \setminus (\mathcal{P} \bowtie \mathcal{J}).
\]

By the allowedness property, the literals used to evaluate $\mathcal{P}$ must include all the variables in $\mathcal{J}$. Let $\chi$ be a subset of the variables in $\mathcal{J}$. Using the identity
\[
    \mathcal{P} = \mathcal{P} \bowtie \mathcal{P}[\chi],
\]
Algorithm 7. The structured bottom-up eval function

```plaintext
function eval (q: query atom, mq: set of magic facts for q)
M_e := M_u \cup mq
repeat
  for each derivation rule R in \text{prules}(\text{pred}(q)) with body B do
    let \mathcal{P} be the extension of the conjunction of positive literals in B
    for each negative literal \neg \epsilon s in the SIPS-induced total order of \mathcal{P} do
      if there is no arc \mathcal{N} \rightarrow \chi \neg \epsilon s then
        \mathcal{P} := \mathcal{P} \setminus \text{ext}(s)
      else
        \mathcal{P} := \mathcal{P} \setminus \text{eval}(s, \text{magic}_s(\text{ext}(\mathcal{N}[\chi])))
      fi
    od
    project on the head variables and obtain \delta R
  od
  for each rule R do add the \delta R to M_e od
until no new facts are added to M_e
return ginsr(q, M_e)
```

The expression (2) can be written as the set difference
\[ \mathcal{P} \setminus ((\mathcal{P} \times \mathcal{P}[\chi]) \bowtie \mathcal{A}), \] (4)

which is equivalent to
\[ \mathcal{P} \setminus (\mathcal{P} \bowtie (\mathcal{P}[\chi] \bowtie \mathcal{A})) \] (5)

by the associativity of \bowtie. The intuition of rewriting the set difference of (2) in this way is similar to that of the semijoin in that we join \mathcal{P} with the smaller relation \((\mathcal{P}[\chi] \bowtie \mathcal{A})\). If \chi contains all the variables of \mathcal{A}, then we don't need the further join with \mathcal{P} in (5) and we perform
\[ \mathcal{P} \setminus (\mathcal{P}[\chi] \bowtie \mathcal{A}). \] (6)

We use this idea in the context of the eval function as follows.

Let \neg \epsilon s be a literal in a rule \textbf{R} whose body is \mathcal{B}. Rather than compute the full extension of \epsilon, we can compute a subset of the extension according to the SIPS we chose. If \mathcal{N} \rightarrow \chi \neg \epsilon s is such an arc, then we can compute the extension of \mathcal{N}, then project it on \chi and compute only that part of the extension of \epsilon that agrees with this projection. This is achieved by making each such tuple in the projection a magic_s fact and invoking the program segment for \epsilon using these magic facts. The magic facts derived in this way are denoted by \text{magic}_s(\text{ext}(\mathcal{N}[\chi])).

The eval function is presented in Algorithm 7. The global variable \text{M}_e denotes the set of ground facts we have deduced. It is initially set to \text{F} before eval is invoked (in an implementation, we wouldn't load the entire set of facts into \text{M}_e; instead, we would use the facts "on demand"). Note that because the DAG is acyclic and there are no function symbols in our databases, the number of calls to eval is finite and each call terminates.
We use the following denotations:

$\delta R$ denotes the new tuples of the extension of the lead predicate of $R$ computed by $R$ during an iteration of $eval$.

ginst$(s, M)$: given an atom $s$ and set of ground atoms $M$, this function returns the set of ground instances of $s$ in $M$.

**REMARKS.**

The magic transformation has already implemented the SIPS leading to positive literals.

The $eval$ function takes care of SIPS arcs leading to negative literals, so that the magic sets for the negative literals are initialized.

For each rule we have used one relation, the extension of the conjunction of all the positive literals in the body. For purposes of exposition this simplifies both the description of the algorithm and its proof of correctness. In practice, implementations may use separate extensions for each positive literal or combination of positive literals.

For simplicity, the $eval$ function has formed the extension of the conjunction of all the positive body literals before removing any tuples corresponding to negative literals. At face value, this may appear to contradict given SIPS. This is not the case, as we demonstrate by the following example. Consider a rule

$qf(X) \leftarrow p^f(X), \neg sbf(X,Y), r^f(Y), t^{bb}(X,Y)$

with SIPS arcs

$\{p^f(X)\} \rightarrow_s s(X,Y)$,

$\{p^f(X), \neg s(X,Y), r(Y)\} \rightarrow_{(X,Y)} t(X,Y)$.

The first arc is straightforward. The second arc states that $\langle X,Y \rangle$ should be evaluated by the conjunction in the tail and passed to $t$. This strategy is implemented by $eval$ faithfully. Note that the strategy does not state anything about the order in which the tuples corresponding to $sbf(X,Y)$ are removed. In order to impose a particular order of computation we could modify $eval$ so that the extensions are evaluated in the desired order. In the presence of a negative literal $\neg s$, however, care must be taken to ensure that $s$ tuples are only removed from an extension which includes all the arguments of $s$ as columns. An alternative approach to modifying $eval$ is to rewrite the rules so that the desired order of computation will occur. In the example above, if we want to compute in the order $(p, \neg s, r), t$ we can rewrite the rules as

$qf(X) \leftarrow temp^{ff}(X,Y), t^{bb}(X,Y)$,

$temp^{ff}(X,Y) \leftarrow p^f(X), \neg sbf(X,Y), r^f(Y)$.

This type of rewriting is entirely analogous to the common subexpression-
elimination optimization of supplementary magic sets and is analogous to creating temporary relations. The point which we wish to stress is that SIPS only dictate what binding information is passed, and the source of this information.

Lemma 1. Let \( D^a \) be an adorned database in which the only negative literals are base literals, and let the adorned query be \( \leftarrow q^a \). Let \( P^m \) be the program transformed by the magic-set transformation of Algorithm 2, so that \( D^m = P^m \cup \text{magic}(q^a) \cup F \). Then \( P^a = \text{magic}^a P^m \).

PROOF. Since the only negative literals are base literals, these cannot be involved in a negative cycle, and so \( P^m \) is stratified. We can show \( P^a = \text{magic}^a P^m \) by incorporating a small change in the proof of Proposition 3 in Section A.1 of the Appendix. In particular, we can consider the height of a negative body literal \( \neg s \) in the context of that proof to be the same as the height of \( s \) and, by the semantics of \( T_i \), strictly less than the height of the head of the rule. The lemma then follows directly from the proof. \( \square \)

Proposition 9. For an adorned, stratified, and allowed database \( D^a \), let \( D^m \) be the magic-transformed database corresponding to \( D^a \), without the magic rules constructed for negative literals. Let \( \leftarrow q^a \) be the adorned query atom, and the model \( M^a \) for \( D^a \) be computed by invoking \( \text{eval}(q^a,\text{Magic}(q^a)) \) where the root of the DAG is \( \mathcal{A}(D^m, \text{pred}(q^a)) \). Then

\[
ginst(q^a, M^a) = ginst(q^a, M(D^a)). \tag{7}
\]

PROOF. We show that Equation (7) holds by induction on the height of each node, where we define the height of a node \( \text{pred}(q^a) \) to be the length of the maximum path in \( \mathcal{A}(D^m, \text{pred}(q^a)) \) from the subtree rooted at \( \text{pred}(q^a) \) to a leaf node.

**Base case:** For a height of 0 there are two possibilities: either \( \text{prules}(\text{pred}(q^a)) \) only contains positive literals or \( \text{prules}(\text{pred}(q^a)) \) also contains negative literals corresponding to base predicates. Using Lemma 1, \( ginst(q^a, M^a) = ginst(q^a, M(D^a)) \), as required.

**Inductive case:** Suppose that the height of \( \text{pred}(q^a) \) is \( i + 1, i \geq 0 \), and that for each negative derived literal \( \neg s \) in \( \text{prules}(\text{pred}(q^a)) \), where the height of the node \( \text{pred}(s) \) is, by definition, less than or equal to \( i \) for any query \( s \theta \), where \( \theta \) simply binds each variable in a bound argument position to a constant

\[
\text{eval}(s, \text{Magic}(s \theta)) = ginst(s \theta, M^a) = ginst(s \theta, M(D^a)).
\]

**Case 1:** There is no arc to \( \neg s \). There are no magic \( s \) rules, since \( s \) has a fully free adornment. By the induction hypothesis, this extension is the extension of \( s \) that would be used in a standard bottom-up computation, based on the \( T_i \) operator. The removal of \( s \) tuples by \( \text{eval} \) is then analogous to the base case.

**Case 2:** There is an arc to \( \neg s \). For each such negative derived literal, it is the second argument in the call to \( \text{eval}, \text{magic}(s(\mathcal{M}[x])) \), which determines \( \theta \) based on the arc \( \mathcal{N} \rightarrow x \rightarrow s \), so that

\[
\text{magic}(s(\mathcal{M}[x])) = \{\text{Magic}(s \theta_1), \ldots, \text{Magic}(s \theta_m)\}.
\]
where each $\theta_j$, $1 \leq j \leq m$, simply binds each variable in a bound argument of $s$ to a constant. The only other reference to the second argument of eval in the initialization of $M_e$, and so, for $m > 1$, the effect of calling eval with $\text{magic}_s(\text{ext}({\mathcal N})(x))$ as its second argument is equivalent to a sequence of calls for which the second argument is $\text{Magic}(s\theta_j)$, $1 \leq j \leq m$. The set of ground instances returned by the successive calls is then identical to the set of ground instances that would be returned had there been just one call. Therefore, for all heights up to and including $i$, the induction hypothesis is

$$\text{eval}(s, \text{magic}_s(\text{ext}({\mathcal N})(x)))$$

$$= \{ \text{ginst}(s, M(D^a)) \text{such that each ground instance of } s$$

$$\text{is an instance of at least one } s\theta_j, 1 \leq j \leq m \}$$

$$= \text{ext}({\mathcal N})(x) \uplus \text{ginst}(s, M(D^a)). \quad (8)$$

For the inductive case, in computing ground instances for the head predicates in a node of height $i + 1$, for each literal $\neg s$, the difference operation performed in eval is

$$\mathcal{P} := \mathcal{P} \setminus \mathcal{P} \uplus \text{eval}(s, \text{magic}_s(\text{ext}({\mathcal N})(x))). \quad (9)$$

Substituting (8) in the right-hand side gives

$$\mathcal{P} \setminus (\mathcal{P} \uplus \text{ext}({\mathcal N})(x) \uplus \text{ginst}(s, M(D^a))).$$

Now, at this point,

$$\mathcal{P} \uplus \text{ext}({\mathcal N})(x) = \mathcal{P}.$$

To see this note that $x$ corresponds to columns in $\mathcal{P}$ and that if $\mathcal{N}$ contains only positive literals then $\text{ext}({\mathcal N})$ is already present in $\mathcal{P}$. If $\mathcal{N}$ contains a negative literal $\neg s'$, then since $s'$ must appear to the left of $s$ in the SIPS-induced total order of the body, $\text{ext}(s')$ must already have been removed from $\mathcal{P}$. Thus, the difference operation performed for each $\neg s$ is

$$\mathcal{P} \setminus (\mathcal{P} \uplus \text{ginst}(s, M(D^a))).$$

Since the extension of $s$ has been computed correctly, the operation is analogous to the base case and Equation (7) holds in the inductive case. \qed

**Proposition 10.** For a stratified and allowed database $D$, let $D^m$ be the magic-set-transformed database corresponding to $D^a$, without including the magic rules constructed for negative literals. Let $\leftarrow q(i)$ be the query to $D$, and let $q^a(i)$ denote the adorned query atom. Let the model $M_e$ for $D^m$ be computed by invoking eval($q^a(i)$, Magic($q^a(i)$)). A ground instance $q^a(\vec{c}) \in M_e$ if and only if $q(\vec{c}) \in M(D)$.

**Proof.** By Proposition 2, $P \equiv^q D^a$. By Proposition 9, a ground instance $q^a(\vec{c})$ is returned by eval($q^a(\vec{c})$, Magic($q^a(\vec{c})$)) if and only if $q^a(\vec{c}) \in M(D^a)$. Therefore, a ground instance $q^a(\vec{c})$ is returned by eval($q^a$, Magic($q^a$)) if and only if $q(\vec{c}) \in M(D)$. \qed
7.5. Domain Independence

The underlying notion behind SIPS, the adornment algorithm, and our definition of allowedness is that they relate to a specific query form as depicted by the adornment. For each different query form, a (potentially) different set of SIPS, adorned database, and magic-set-transformed database is generated. Following this line, we show that the property of domain independence that allowed databases possess is also query- and SIPS-dependent.

Definition 32. A database $D$ is domain-independent with respect to a query $\leftarrow q$ if the set of instances of $q$ in $M(D)$ does not change when the language of $D$ is extended.

Proposition 11. If the adorned database $D^a$ resulting from a query $q$ on a stratified database $D$ with SIPS $S$ is allowed, then $D$ is domain-independent with respect to $q$.

Proof. For each $D^a$ with allowed SIPS and query $q$, there exists a corresponding $D^m$. By Proposition 9, $P^m$ is equivalent to $P^a$ with respect to $q$. Consider now the rules in each node of the DAG for $P^m$. By Proposition 6 the rules in each node are Clark-allowed and therefore domain-independent [45]. If it were possible for a new constant to be added to the language associated with $D^m$, then $eval$ must somehow have returned this new constant during query evaluation. To refute this possibility, we proceed, again, using a proof of induction on the height of a node.

Base step: Clearly for a height of 0, $eval$ does not introduce any new constants, since the computation of ground instances of a predicate corresponding to a negative literal in the Clark-allowed $prules(pred(q))$ involves operations over base relations. Similarly, when there is no negative literal, since the rules are Clark-allowed, they are domain-independent.

Inductive step: Suppose the height is greater than 0, and for each negative literal $\neg s$ in $prules(pred(q))$, for any query $s\theta$, where $\theta$ simply binds variable in a bound argument position of $s$ to a constant,

$$eval(s, Magic(s\theta)) = ginst(s\theta, M(D^a)).$$

We now analyse the source of $\theta$. Let $N \rightarrow x \neg s$ be an arc from the SIPS. When $N$ does not contain negative literals, $eval$ computes the extension of $N$ and projects on $x$. Now the extension of $N$ involves only positive literals which are connected to $x$, by the definition of allowed SIPS, and so this extension cannot generate a $\theta$ which introduces a new constant. If $N$ does contain a negative literal, then according to the total order induced by the SIPS we will have computed the extension for this negative literal. The definition of allowed SIPS implies that every argument of a negative literal in $N$ must also appear in either a positive literal in $N$ or the bound argument position of the head $p^a \in N$. Equivalently, projecting $x$ over the extension of $N$ and recasting the resultant tuples as $Magic(s)$ facts is identical to using the magic rule constructed for $\neg s$. All magic rules are allowed by Proposition 6, and thus each $\theta$ involves only constants in the language of $D^m$. Evaluating the extension of the body of a rule must therefore involve algebraic operations over constants in the language, and so the rules in $pred(q)$ are domain-independent. Thus, $D^m$ is domain-independent with respect to $q$. Since $P^a$
is equivalent to $P^m$ with respect to $q$, $D^a$ is domain-independent with respect to $q$ as well. □

Recall that although $D^m$ may be domain-independent for disallowed SIPS, there is a practical motivation for restricting ourselves to allowed SIPS, as discussed in Section 5.3, and thereby to a Clark-allowed $D^m$.

7.6. System Overview

The system we propose for using magic-set methods when a query $\leftarrow q$ is given to a database $D$ is divided into the following stages. Note that all stages but the last are done at compile time.

1. Check that $D$ is stratified by looking for negative cycles in the dependency graph of $D$. The system reports an error if a negative cycle is found.
2. Apply adorn to $D$, $S$, and $q$ to give $D^a$ and $q^a$ and $S^a$.
3. Check that all rules and SIPS in $D^a$ and $S^a$ are allowed. The system reports an error if any rules or SIPS are not allowed.
4. Apply the magic-set transformation of Algorithm 3 to $D^a$ and $S^a$ to give $D^m$.
5. Construct the negation-induced graph $\mathcal{G}(D^m, \text{pred}(q^a))$.
6. Output the set of ground atoms returned by $\text{eval}(q^a, \text{Magic}(q^a))$.

In evaluating the initial tuples for a derived predicate $p$ we normally load any relevant facts from the database. This initialization can be extended to any rule defining $p$ that does not contain a body literal that depends on a recursive predicate. By a process of unfolding, a relational expression is obtained solely in terms of base predicates for initializing the set of $p$ tuples. This saves much unnecessary computation, since the body of those rules is evaluated once only. Indeed, whenever it is feasible to unfold, it will be more efficient to do so before step 2.

8. RELATED WORK

In this section we describe related work. Before we do so, we point out that top-down or hybrid top-down–bottom-up algorithms such as those described in [49, 24] do not suffer from the stratification and allowedness problems described in this paper. The primary reason for this is that they do not transform the database and therefore do not have to solve the “mixing of contexts” problem.

The first paper to describe a magic-set algorithm for normal databases, the GMS algorithm, was [11]. Comments with reference to this work have been made throughout the present paper. Unstratification, however, was not seen to pose a major problem in [11]. The authors did not offer a formal description of the specialized control required to answer queries to normal databases consistent with the perfect model semantics. Our first description of the solution to this problem
An improved labeling algorithm to accompany the work in [4] was published in [3]. Subsequently, research describing another specialized control strategy was done by the Alexander group, whose work was published in [25]. The approach taken there differs from ours. We sought to focus on the unstratification and discover the causes as a means of solving the problem. They defined a new model, called the W-model. The W-model is not touted as a new semantics for normal databases. Instead, the authors claim that a ground instance of the query is in the W-model of the Alexander-transformed database if and only if this ground instance is in the perfect model of the original database.

The computation of the W-model can be informally described as follows. Partition the rules in the database into two distinct sets: those rules which contain negative body literals and those which do not. Next, divide each set into appropriate strata. The computation of the W-model begins with a bottom-up computation of the set of rules that contain only positive body literals. When this has terminated, any newly derived ground instances are available for a separate bottom-up computation of the rules that contain negative literals. After this computation has terminated, we revert back to the previous set of rules, sharing any newly derived ground instances, and repeating the process until no more new ground instances can be derived by a computation of either set of rules.

The Alexander scheme outlined above has some parallels with the method employed by our $eval$ function. Effectively, there are two contexts in which rules are used. One is the positive case (no negative body literals), and the other the negative case. They are separated by dividing the database and toggling between fixpoint computations of the rules in each case. Thus, when answering a negative literal $\neg s$, any new $magic_s$ tuples would have been derived in the previous fixpoint computation of the set of rules which do not contain negative literals.

9. CONCLUSIONS

In the past, magic sets formed the basis of an efficient bottom-up query processing technique on positive deductive databases. In this paper, we have extended the scope of the magic-set algorithm to include normal deductive databases. Before extending the magic-set algorithm, we presented a simplified and improved version of the algorithm for positive deductive databases. We then defined a new, less restrictive definition of allowedness and introduced the concept of allowed SIPS. We proved that when the magic-set algorithm is applied to an allowed adorned database with allowed SIPS, the resultant database is allowed. When the magic-set algorithm is applied to normal databases, we showed that the resultant database is often unstratified. In developing our solution to this problem, we built on the intuition of the labeling approach [3] and presented a new method to efficiently answer a query using magic sets in accordance with the perfect model semantics. The solution involved a modification to the standard bottom-up procedure, using the negation-induced graph. The net result is an efficient, correct, and domain-independent method for answering a query on a normal stratified database based on a bottom-up computation using magic sets.
APPENDIX. PROOFS

A.1. Magic-Set Transformation—Algorithm 2

In this proof we make use of the following definition, which makes use of definitions from Section 4.

**Definition 33.** Let \( p^g \) denote a ground atom appearing in \( g(D) \), and \( R_g \) denote a ground rule in \( g(D) \). Define

\[
\text{height}(p^g) = 1 + \min \{ \text{height}(R_g) \mid R_g \text{ defines } p^g \},
\]

\[
\text{height}(R_g) = \begin{cases} 
0 & \text{if } R_g \text{ is a fact,} \\
\max \{ \text{height}(p_i^g) \mid p_i^g \text{ is a body literal in } R_g \} & \text{otherwise.}
\end{cases}
\]

As evidenced by the definition, the height of a ground atom in \( M(D) \) is the minimum number of iterations or applications of the \( T \) mapping required until that atom appears as a ground instance of a rule of \( g(D) \).

**Proposition 3.** For a positive adorned database \( D^a \) and adorned query \( q^a \), let \( P^m \) be the program transformed by the magic-set transformation of Algorithm 2, so that \( D^m = P^m \cup \text{Magic}(q^a) \cup F \). Then \( P^a \equiv_q P^m \).

**Proof.** Set of instances of \( q^a \) in \( M(D^a) \) \( \subseteq \) set of instances of \( q^a \) in \( M(D^m) \). Let \( g^{q^a}(D^a) \) be the ground relevant rule set corresponding to \( q^a \) and \( D^a \), and let \( H = \text{head}(g^{q^a}(D^a)) \). By definition, the set of instances of \( q^a \) in \( M(D^a) \) is a subset of \( H \). We show that \( H \subseteq M(D^m) \). This is done by induction over the elements of \( H \) on the height of each element. We prove that for each \( p^g \in H \),

\[
p^g \in M(D^{m+p}), \quad \text{where } D^{m+p} = D^m \cup \{ \text{Magic}(p^g) \}. \tag{10}
\]

This is sufficient to show that \( H \subseteq M(D^m) \), since \( \text{Magic}(q^a) \) is included in \( D^m \) by the magic-set algorithm and \( H \) consists solely of ground atoms relevant to the instances of \( q^a \) in \( M(D^m) \).

**Base case:** For each \( p^g \in H \) such that \( \text{height}(p^g) = 1 \), \( p^g \) is a fact in \( D^a \) and so \( p^g \) is in \( M(D^{m+p}) \).

**Inductive case:** Suppose that for some \( n \), for all \( i < n \), and for each ground atom \( p^g \in H \) of height \( i \), \( p^g \) is also in \( M(D^{m+p}) \). We prove that a ground atom \( p^g_0 \in H \) of height \( n \) is also in \( M(D^{m+p}) \). Since \( p^g_0 \) is in \( H \), so that \( \text{height}(p^g_0) = n \), \( n > 1 \), there must a rule \( R \) in \( D^a \) with head \( p_0 \) and body \( p_1, \ldots, p_m \), \( m \geq 1 \), such that \( R_g \) is an instance of \( R \) and the head of \( R_g \) is \( p^g_0 \). The magic-set algorithm transforms \( R \) so there is a rule of the form

\[
p_0 \leftarrow \text{Magic}(p^g_0), p_1, \ldots, p_m
\]

in \( D^{m+p} \). For \( p^g_0 \) to be in \( M(D^{m+p}) \),

\[
p^g_0 \leftarrow \text{Magic}(p^g_0), p^g_1, \ldots, p^g_m
\]

must be a ground instance of this rule. Consider the ground body literals. Since \( \text{Magic}(p^g_0) \) is included in \( D^{m+p} \), all we need to show is that each ground instance
According to the induction hypothesis, each \( p_j^g \) is in \( H \), since, by the definition of the height function, the \( p_j^g \) are of height \( i \) such that \( i < n \). To show that each \( p_j^g \) is in \( M(D^{m+p}) \) we require to prove that \( \text{Magic}(p_j^g) \) is in \( M(D^{m+p}) \) also.

We show that each such \( \text{Magic}(p_j^g) \) is in \( M(D^{m+p}) \) by considering the derived body literals in the total order induced by the SIPS associated with \( R \). Let \( p_i \) be the first derived literal in the body. There is an arc \( \mathcal{N} \rightarrow p_i \) from the SIPS, in which \( \mathcal{N} \) consists of base literals and possibly \( p_0 \). So there is a magic rule with head \( \text{Magic}(p_i) \) and body consisting of the base literals in \( \mathcal{N} \) and possibly \( \text{Magic}(p_0) \). Since the facts defining the base predicates in \( D^a \) are included in \( D^{m+p} \) and \( \text{Magic}(p_0^g) \) is in \( D^{m+p} \), then \( \text{Magic}(p_j^g) \) is in \( M(D^{m+p}) \). By the induction hypothesis this implies that \( p_j^g \) is in \( M(D^{m+p}) \). Consider the next derived literal \( p_2 \) in the SIPS-induced order. The head of the SIPS arc \( \mathcal{N} \) entering \( p_2 \) may include \( p_1 \), and the corresponding magic rule would include \( p_1 \) in the body. Since \( p_1^g \) is in \( M(D^{m+p}) \), using a similar argument to the above we can show that \( \text{Magic}(p_1^g) \) and \( p_2^g \) are in \( M(D^{m+p}) \). Repeating this for all such derived body literals, we conclude that Equation (10) holds for this \( p_0^g \) as well.

By induction, the hypothesis holds for each member of \( H \). As mentioned, since \( \text{Magic}(q^a) \in D^m \), it follows that \( H \subseteq M(D^m) \), which completes the proof in this direction.

**Set of instances of \( q^a \) in \( M(D^m) \; \subseteq \text{set of instances of } q^a \text{ in } M(D^m) \):** The proof in this direction simply follows from the fact that rules in \( D^m \), which are derived from rules in \( D^a \), are more restrictive in that an extra positive body is inserted into the body. □

### A.2. Labeling Transformation—Algorithm 4

Let us focus on the properties of the positive labeling procedure at the point marked (*) in Algorithm 6. The \( i \) corresponds to the stratum \( I_i \) defining a negatively labeled predicate \( n_s \). At iteration \( j \) of the procedure we positively label each unlabeled derived predicate \( p \) appearing in \( I_j \) such that \( n_s \leftarrow p \). The predicate \( p \) is renamed as \( p_k \). The integer \( k \) is the value of the counter \( C_j \). At (*), for each value of \( j \) every such predicate \( p \) has exactly one common (negatively labeled) ancestor predicate \( n_s \) defined in \( I_i \). Each such predicate \( p \) will therefore be given the same value of \( k \) as its positive label. We denote this invariant property by the relation

\[
\text{neg_parent}(j, k) = i,
\]

where \( k = C_j \). That is, for each negatively labeled predicate \( n_s \) defined in \( i \), each positive derived predicate defined in a stratum \( j \) (\( j < i \)) will be given the same positive label \( k(p, k) \). We shall use this invariant and the following definition (incorporating a small modification from the original definition in [20]) for proof purpose.

**Definition 34.** The condensation \( \mathcal{G}^* \) is a directed graph derived from the MSCCs of \( \mathcal{I} \). Each node \( c_i \) in \( \mathcal{G}^* \) corresponds to a MSCC in \( \mathcal{I} \). We create an arc \((c_i, c_j)\) between two nodes in \( \mathcal{G}^* \) if there is an arc \((m_i, m_j)\) in \( \mathcal{I} \) such that the
predicate $m_i$ is a member of the component $c_i$, and the predicate $m_j$ is a
member of $c_j$. If a positive (negative) arc $(m_i, m_j)$ exists, then the arc $(c_i, c_j)$
exists, and is correspondingly positive (negative).

By construction, the following properties pertain to the condensation graph $G^*$.

H1. Each MSCC exclusively contains either negatively labeled predicates, positively labeled predicates, or unlabeled predicates.

H2. Each stratum may contain one MSCC of negatively labeled predicates (which we denote by $n_i$), one MSCC of unlabeled predicates (which we denote by $u_i$), and $k$ MSCCs of positively labeled predicates $p_k$ (which we denote by $p_k$).

H3. All negative arcs originate from MSCCs of negatively labeled predicates and are of the form $(n_i, c_j)$ where $j > i$ and $c_j$ is a node whose predicates are defined in stratum $j$.

H4. Positive arcs that originate from an MSCC of negatively labeled predicates are loop arcs (that is, the arc is not incident on a different MSCC).

H5. Consider a positive arc originating from an MSCC of positively labeled predicates $p_k$ to another node $c_i$. Either
   (1) $c_i$ is $p_k$, and thus it is a loop arc; or
   (2) $c_i$ is an MSCC of negatively labeled predicates $n_i$, where neg_parent $(j, k) = i$; or
   (3) $c_i$ is $p_m$, such that neg_parent $(j, k) = neg_parent(i, m)$.

H6. Arcs that originate from an MSCC of unlabeled predicates are positive and are incident only on an MSCC of unlabeled predicates.

Lemma 2. Let $D$ be any initial stratified and allowed database with associated program $P$, and let $\leftarrow q(i)$ be a query. Let $D'$ be the corresponding labeled database and $P'$ the adorned program, and let $\leftarrow q^a(i)$ be the adorned query. The labeled database $D'$ is stratified and allowed.

Proof. Allowedness is clearly not affected by the renaming of predicates, and so the labeling algorithm preserves the allowedness of the original program. To show that stratification is preserved we proceed with a proof by contradiction. Assume that $P'$ contains a negative cycle and that $P$ does not. By H3 and H1 the negative cycle begins and ends with a negatively labeled predicate and all predicates in the cycle are either positively labeled or negatively labeled. If we drop all the labels in the cycle, then by construction of the algorithm, the cycle (with labels removed) also exists in $P^a$. To see this note that if we drop the labels from any rule in $P'$, the resultant rule must also be in $P'$. The rules in $P^a$ are stratified, and so we derive a contradiction and conclude that $P'$ is stratified also. □

Let $D''$ be the database obtained after applying the magic-set algorithm (on positive body literals) to $D'$, and let $S''$ be the associated dependency graph. Based on $S''$, we construct an abstraction of $S''$ that we denote by $S'''$ and that is different to the condensation $S^{m*}$. Each node $c'_i$ in $S'''$ is identical to the
corresponding node $c_i$ in $G^{1*}$, except that the predicates constituting the MSCC $c_i$ also include associated magic predicates. Each arc $(c_i, c_k) \in G^{1*}$ exists in $G^m$ as the arc $(c_i', c_k')$. In addition, if the arc $(c_i, c_k)$ is positive, and the MSCC associated with $c_i$ contains derived predicates, then $G^m$ has a reversed positive arc $(c_k', c_i')$.

The pseudograph $G^m$ is an abstraction of the dependency graph $G^m$ in the following sense.

**Lemma 3.** Let $D^m$ be the resultant database after applying the magic set transformation (of positive body literals) to the labeled database $D^l$. If there is a negative cycle in the dependency graph $G^m$ of $D^m$, then there is a negative cycle in $G^m$.

**PROOF.** Consider the application of the magic-set algorithm to a rule in $D^l$

$$h \leftarrow p_1, \ldots, p_n, \neg s_1, \ldots, \neg s_m,$$

where $n, m \geq 0$ and where the order of the body literals is immaterial to the proof. For notational convenience, we denote the node of a graph by any predicate associated with that node. Depending upon the SIPS associated with the rule, the following arcs may appear in $G^m$: $(\text{magic}_h, h), (\text{magic}_h, \text{magic}_p), (p_k, \text{magic}_p), (-s_j, \text{magic}_p)$, where $i, k \leq n$ and $j \leq m$. We now analyse these arcs and show that if any of them is a member of a negative cycle in $G^m$, then a corresponding path exists in $G^m$.

1. $(\text{magic}_h, h)$: Since $\text{magic}_h$ and $h$ share the same node in $G^m$, this is a loop arc in $G^m$ that is, an arc of the form $(c, c)$. Furthermore, since a negative cycle will still pass through the node whose MSCC includes $h$, independently of loop arcs, there is no need to include this arc in $G^m$ explicitly.

2. $(\text{magic}_h, \text{magic}_p)$: This type of arc is in the reverse direction to the existing positive arcs $(p_i, h)$ of $G^m$, and so it is a member of $G^m$ by construction.

3. $(p_i, \text{magic}_p)$ and $(-s_j, \text{magic}_p)$: For these arcs, the corresponding paths

$$\left(p_i, h\right), (\text{magic}_h, \text{magic}_p),$$

and

$$\left(-s_j, h\right), (\text{magic}_h, \text{magic}_p)$$

already exist in $G^m$, and so it is not necessary to include them in $G^m$ explicitly.

Since all the arcs $(p_i, h)$ and $(s_j, h)$ are in $G^m$, if there is a negative cycle in $G^m$, then there is a negative cycle in $G^m$. □

The following properties pertain to the abstraction $G^m$:

**A1.** The only negative arcs in $G^m$ are those in $G^{1*}$, since there are no reverse negative arcs in $G^m$. Any negative cycle must pass through a node $n'_i$. Since $D^l$ is stratified, these arcs are of the form $(n'_i, c'_j)$ where $j > i$. Therefore, since there are no negative loop arcs in $G^{1*}$, there are none in $G^m$. 
A2. The only positive arcs originating from \( n_i \) are loop arcs and reverse arcs to \( p_k' \) such that \( \text{neg}_\text{parent}(j, k) = i \).

A3. At each level \( i \) there is at most one \( n_i \). If we let \( \varphi, 1 \leq \varphi \leq n \), denote the highest level for which there exists an MSCC of negatively labeled predicates, then for any positively labeled predicate \( p_k', j < \varphi \).

In proving that there are no negative cycles in \( \mathcal{G}^m \) we use the following lemmas.

**Lemma 4.** No MSCC of unlabeled predicates \( u_i' \) is part of a negative cycle.

**Proof.** By property H6, since the only arcs originating from unlabeled predicates \( u_i' \) are to MSCCs of unlabeled predicates, no such path exists. \( \Box \)

**Lemma 5.** If \( p_k' \) is part of a negative cycle, then that cycle must pass through \( n_i \), where \( i = \text{neg}_\text{parent}(j, k) \).

**Proof.** Let there be a negative cycle through \( p_k' \). This must pass through a \( n_m' \). If \( m = i \), the lemma is proved. Otherwise \( n_i \) is on a path from \( p_k' \) to \( n_m' \) by H4 and H5(3). Therefore, \( n_i \) must also be part of the cycle. \( \Box \)

**Lemma 6.** If \( n_i \) is part of a negative cycle, then a negative arc originating from \( n_i \) is part of the cycle.

**Proof.** We show that no positive arc from \( n_i \) is involved in a negative cycle. By property A2, for a negative cycle we need only consider arcs to \( p_k' \) such that \( \text{neg}_\text{parent}(j, k) = i \). Consider a negative cycle involving one such positive arc \((n_i', p_k')\). Since this arc is positive, there must be a negative arc in the path from \( p_k' \) to \( n_i' \). The only possible paths from \( p_k' \) to \( n_i' \) pass through nodes \( p_{h'} \) such that \( \text{neg}_\text{parent}(h, t) = i \). Since there are no negative arcs from positively labeled nodes, there is no negative arc in this path. Hence no positive arc from \( n_i \) is involved in a negative cycle. \( \Box \)

**Proposition 8.** If \( D' \) is the resultant database after applying the labeling transformation of Algorithm 4 to a stratified database \( D \), and \( D^m \) is the resultant database after applying the magic-set transformation of Algorithm 3 (constructing magic rules only for positive literals) to \( D' \), then \( D^m \) is stratified.

**Proof.** We show that there is no negative cycle in \( \mathcal{G}^m \). Let \((c_i', c_j')\) be a negative arc in \( \mathcal{G}^m \). Clearly, the predicates associated with \( c_i' \) are negatively labeled. That is, \( c_i' = n_i' \). By property A1, there cannot be a negative loop arc (that is, \( i \neq j \)). We show by induction that no \( n_i' \) in \( \mathcal{G}^m \) is part of a negative cycle. By Lemma 6, we need only consider negative arcs originating from \( n_i' \).

Let \( n_{\varphi} \) denote the highest level for which there exists an MSCC of negatively labeled predicates.

**Basis:** \( n_{\varphi} \) is not involved in a negative cycle. Let \((n_{\varphi}', c_i')\) be the negative arc from \( n_{\varphi}' \) that is part of the negative cycle. We show that no such \( c_i' \) can exist. Since \( j \) is greater than \( \varphi \), by property A3, \( c_j' \) cannot be a \( p_k' \) or a \( n_i \) and so \( c_j' \) must be a \( u_i' \). However, by Lemma 4, no \( u_i' \) can be part of a negative cycle, and therefore \( n_{\varphi} \) is not involved in a negative cycle.
**Induction statement:** If no $n'_i$ at level $s > i$ is part of a negative cycle, then neither is $n'_j$. Let $(n'_i, c'_j)$ be the negative arc that is part of a negative cycle. We show that no such $c'_j$ exists. By Lemma 3, if $c'_j$ is a $u'_j$ it cannot be part of a negative cycle. If $c'_j$ is $n'_j$, then by the induction hypotheses, since $j > i$, it cannot be involved in a negative cycle. If $c'_j$ is $p_{-k}'$, then there must exist an $n'_m$ such that $\text{neg} \_ \text{parent}(j, k) = m$. Since $m > j$ and $j > i$, we have $m > i$. By Lemma 5 and the induction hypotheses $p_{-k}'$ is also not part of a negative cycle. Thus, no negative arc from $n'_i$ is part of a negative cycle, and so $\mathcal{G}'^m$ has no negative cycle, and, by Lemma 3, $\mathcal{D}'^m$ is stratified.

We would like to thank the anonymous referees for their detailed comments on an earlier draft. Special thanks are due to David Kemp and Zoltan Somogyi for their careful reading of this paper. This research was partially supported by a University of Melbourne Postgraduate Research Award, a Commonwealth Postgraduate Research Award, and a grant from the Australian Research Council for the Machine Intelligence Project at the University of Melbourne.

**REFERENCES**


42. Sagiv, Y., Optimizing Datalog Programs, in: [32], pp. 659–698.


