Constraint Stratification in Deductive Databases

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

Abstract

We propose a syntactic condition on deductive database programs that ensures a two-valued well-founded model. This condition, called constraint stratification, is significantly more general than previous syntactic conditions such as stratification and local stratification. Modular stratification has been proposed as a semantic (i.e., nonsyntactic) condition for ensuring a two-valued well-founded model. While not every modularly stratified program is constraint stratified, all of the well-known practical examples of modularly stratified programs are constraint stratified under appropriate natural constraints. In addition, there exist constraint stratified programs that are not modularly stratified. We also show how the magic sets optimization technique can be applied for constraint stratified programs.

1 Introduction

Much recent work has concerned defining the semantics of negation in deductive databases. The "perfect model semantics" [Prz88] has been generally accepted as natural, and is the basis for several experimental deductive database systems. Unfortunately, the perfect model semantics applies only to programs that are stratified (or locally stratified). A stratified program is one in which, effectively, there is no predicate that depends negatively on itself.

Recent work has shown that there are interesting logic programs that are not stratifiable but for which a natural, unambiguous semantics exists. The well-founded semantics [VGRS91] and the stable model semantics [GL88] are two (closely related) proposals for defining the semantics of logic programs, whether stratified or not. For stratified programs they both coincide with the perfect model semantics.

The well-founded semantics is a three-valued semantics. Literals may be *true*, *false* or *undefined*. The stable model semantics is also a three-valued semantics in the sense that the meaning of the program is, in general, determined by a *set* of (two-valued) models rather than a single model. However, there are many cases where a non-stratified program has a total semantics, i.e., a semantics in which every ground literal is either true or false. Allowing programs that have some literals *undefined* may not be desirable, since handling this extra truth value places an extra burden on the query evaluation procedure. In many cases, two truth values suffice to model the situation under consideration. So we desire a condition on the program, more general than stratification, that ensures that the well-founded semantics is two-valued.

In [Ros90] the present author proposed such a class, which was termed the class of *modularly stratified* programs. For modularly stratified programs the well-founded semantics is total (i.e., makes every ground literal either true or false). The well-founded semantics and the stable model

*The research of Kenneth Ross was supported by NSF grants IRI-9209029 and CDA-90-24735, by a grant from the AT&T Foundation, by a Sloan Foundation Fellowship, and by a David and Lucile Packard Foundation Fellowship in Science and Engineering.

semantics coincide for modularly stratified programs, a consequence of the fact that the well-founded model is total. Modularly stratified programs also allow subgoal-at-a-time evaluation [Ros90]. A program is *modularly stratified* if and only if its mutually recursive components are *locally stratified* once all instantiated rules with a false subgoal that is defined in a "lower" component are removed.

Unfortunately, the definition of modular stratification is *semantic* rather than *syntactic*. Whether a program is modularly stratified depends, in general, on the semantics assigned to its predicates. This contrasts with stratification and local stratification, which are syntactically definable conditions. Other conditions such as weak stratification [PP90] are also semantic in this sense.

In the context of deductive databases, where the rules and schema-level information is small compared to the data, it would be undesirable to make the stratification condition depend on the data. We should try to come up with a form of stratification that can be defined using only the program itself and some schema-level information.

In this paper we attempt to define a condition that is more general than local stratification, but which can be defined syntactically, without computing the semantics of the program "along the way." In order to do so, we allow the programmer to specify some schema-level constraints on the EDB predicates. The constraints we allow are monotonicity constraints [BS89]. Monotonicity constraints specify that one argument of a predicate is less than another according to some partial order.

Our approach can be outlined as follows:

- Specify a set of monotonicity constraints on the EDB predicates. Such constraints are often natural, restricting an EDB relation to be acyclic, for example, when it represents a part/subpart hierarchy.
- Infer new monotonicity constraints on the IDB predicates using a sound inference mechanism.
- Delete from the instantiated program any rules that have a subgoal violating the constraints, and require that the resulting program be locally stratified.

We need to infer new monotonicity constraints for programs with negation. Brodsky and Sagiv give an inference method for datalog programs without negation. We can adapt their inference method to programs P with negation by deriving constraints for the *envelope* of P, i.e., P with all negative subgoals removed. When the IDB of P function-free, this inference mechanism applies. We prove that it is sound with respect to the well-founded semantics of P.

Not every modularly stratified program is constraint stratified. However, we find, perhaps surprisingly, that all of the common examples of modularly stratified programs, including all of the examples from [Ros90], are constraint stratified. Further, there are some constraint stratified programs that are not modularly stratified.

We prove that constraint stratified programs have a two-valued well-founded semantics. We also outline how magic sets techniques for modularly stratified programs can be applied to a version of constraint stratification that ensures freedom from recursive loops through negation using a left-to-right binding passing strategy.

2 Terminology

We consider normal logic programs with function symbols and negation [Llo87].

Definition 2.1: A *term* is either a variable, a constant symbol, or a function symbol applied to terms. If p is an n-ary predicate symbol and $t_1, \cdots, t_n$ are terms then $p(t_1, \cdots, t_n)$ is an *atom*. A *literal* is either an atom or a negated atom. When we write an atom $p(\vec{X})$ it is understood that $\vec{X}$ is a vector of terms, not necessarily variables.

A *rule* is a sentence of the form

$$A \leftarrow L_1, \ldots, L_n$$

where A is an atom, and $L_1, \ldots, L_n$ are literals. We refer to A as the *head* of the rule and $L_1, \ldots, L_n$ as the *body* of the rule. Each L_i is a *subgoal* of the rule. All variables are assumed to be universally quantified at the front of the rule, and the commas in the body denote conjunction. If the body of a rule is empty then we may refer to the rule as a *fact*, and omit the "$\leftarrow$" symbol. A *program* is a set of rules. □

Logical variables begin with a capital letter; constants, functions, and predicates begin with a lowercase letter. The word *ground* is used as a synonym for "variable-free."

If a predicate is defined only by facts, then we say that the predicate is an *extensional database* (EDB) predicate; otherwise the predicate is an *intensional database* (IDB) predicate.

We shall also make the assumption that programs are *range restricted*, i.e., every variable occurring in the head of a rule or in a negative literal in the body also occurs in a positive literal in the body. Such programs have also been called *allowed* or *safe*.

We assume that a universe $\mathcal{U}$ is given. $\mathcal{U}$ should contain all ground terms that can appear in all possible programs and EDB relations. In particular, $\mathcal{U}$ will include the Herbrand universe of any IDB/EDB pair. $\mathcal{U}$ will function as the domain under consideration, with terms interpreted freely. When we talk about "instantiated" atoms and rules, we mean that values from $\mathcal{U}$ are substituted for all variables in the atom or rule.

A program is *stratified* if there is an assignment of ordinal levels to *predicates* such that whenever a predicate appears negatively in the body of a rule, the predicate in the head of that rule is of strictly higher level, and whenever a predicate appears positively in the body of a rule, the predicate in the head has at least that level.

A program is *locally stratified* if there is an assignment of ordinal levels to *ground atoms* such that whenever a ground atom appears negatively in the body of an instantiated rule, the head of that rule is of strictly higher level, and whenever a ground atom appears positively in the body of an instantiated rule, the atom in the head has at least that level.

We say a predicate p *depends upon* a predicate q if there is a sequence of rules $r_0, \ldots, r_{n-1}$ with predicates $p_0, \ldots, p_{n-1}$ in the head, respectively, such that

1. $p = p_0$ and $q = p_n$, and
2. for $i = 1, \ldots, n$, p_i appears (positively or negatively) in the body of r_{i-1}.

We say p depends on q through k negations if exactly k of the appearances of $p_1, p_2, \ldots, p_n$ in $r_0, \ldots, r_{n-1}$, respectively, are negative. We say p depends *negatively* on q if p depends on q through at least one negation. A predicate p is *mutually recursive with* a predicate q if p depends upon q and q depends upon p.

Let F be a component (i.e., a subset of the rules) of a logic program P. We say F is a *complete* component if for every predicate p appearing in the head of a rule in F,

- all rules in P with head p are in F, and
- if p is mutually recursive with a predicate q, then all rules in P with head q are in F.

If the predicate p appears in the head of a rule in F then we say p *belongs to* F. If the predicate q appears in the body of a rule in F, but does not belong to F, then we say q is *used by* F. If an atom A has predicate p, and p belongs to F, then we may say that A also belongs to F.

If we say that predicates are mutually recursive with themselves, then mutual recursiveness is an equivalence relation between predicates. Every predicate has a unique minimal complete component

to which it belongs. A program may be broken up into complete components according to the equivalence classes (called strongly connected components in [Ull89]) induced on the predicates.

The minimal complete components have a natural relation associated with them: $F_1 \sqsubset F_2$ if some predicate belonging to F_1 is used by F_2. $\sqsubset$ must be an acyclic relation, since if $F_1 \sqsubset F_2 \sqsubset \cdots \sqsubset F_n \sqsubset F_1$ for some n, then none of $F_1, \ldots, F_n$ would be complete. We refer to $\sqsubset^*$, the transitive closure of $\sqsubset$, as the *dependency relation* between components. $\sqsubset^*$ is a partial order, with the property that a predicate belonging to a component F is defined in terms of predicates that either belong to F, or belong to a component F' where $F' \sqsubset^* F$.

In what follows, when we refer to a component of a program, we mean a minimal complete component unless otherwise noted. Within this framework, a program is stratified if and only if none of its components contains a predicate that depends negatively on itself.

Definition 2.2: The *envelope* of a program P is P with all negative subgoals removed. □

2.1 Modular Stratification

We now present the concept of modular stratification, originally defined in [Ros90].

Definition 2.3: (Reduction of a component) Let F be a program component, and let S be the set of predicates used by F. Let M be a two-valued interpretation over the universe $\mathcal{U}$ for the predicates in S.

Form $I_{\mathcal{U}}(F)$, the instantiation of F with respect to $\mathcal{U}$, by substituting terms from $\mathcal{U}$ for all variables in the rules of F in every possible way. Delete from $I_{\mathcal{U}}(F)$ all rules having a subgoal Q whose predicate in S, but for which Q is false in M. From the remaining rules, delete all (both positive and negative) subgoals having predicates in S (these subgoals must be true in M) to leave a set of instantiated rules $R_M(F)$. We call $R_M(F)$ the *reduction of F modulo M*. □

Definition 2.4: (Modular Stratification) We say the program P is *modularly stratified* if, for every component F of P,

1. There is a total well-founded model M for the union of all components $F' \sqsubset^* F$, and
2. The reduction of F modulo M is locally stratified. □

Theorem 2.1: ([Ros90]) Every modularly stratified program has a total well-founded model that is its unique stable model. ▮

To see how the well-founded model of a program may be composed from those of its components, recall that a locally stratified program has a unique perfect model [Prz88] and hence a total well-founded model that coincides with the perfect model. The "lowest" components must be locally stratified; compute their perfect model M. The next lowest components are locally stratified when reduced modulo M; compute the perfect model of the reduced components and take the union with M. We can proceed in this way up the dependency relation between components until we have the well-founded model for the whole program.

2.2 Examples

We present a number of examples from [Ros90] of modularly stratified programs. Note that none of these examples is locally stratified.

Example 2.1: Consider the program P consisting of the rule

$$w(X) \leftarrow m(X,Y), \neg w(Y)$$

together with some facts about m. P is a game-playing program [Kol91] in which a position X is "winning" $[w(X)]$ if there is a move from X to a position Y $[m(X,Y)]$ and Y is a losing position $[\neg w(Y)]$.

P is modularly stratified when m is acyclic, i.e., when the game cannot have repeated positions. □

Example 2.2: This example concerns the operation of a complex mechanism that is constructed from a number of components, each of which may itself have smaller components. We adopt the convention that a mechanism is not a component of itself — we are only interested in smaller, simpler components. The mechanism is known to be *working* either if it has been (successfully) *tested*, or if all its components (assuming it has at least one component) are known to be *working*. We may express this in the following component F:

$$\begin{aligned} &working(X) \leftarrow tested(X) \\ &working(X) \leftarrow part(X,Y), \neg has\_suspect\_part(X) \\ &has\_suspect\_part(X) \leftarrow part(X,Y), \neg working(Y) \end{aligned}$$

Let M be the least model of the rules for *part* and *tested*. $R_M(F)$ is locally stratified if and only if *part* is acyclic. Acyclicity is a natural constraint, since a mechanism that was a sub-part of itself would presumably indicate a design error. □

Modular stratification can be extended to aggregation and set-grouping.

Example 2.3: Suppose we have a relation $part(X,Y,N)$ that is true when X has N copies of Y as an immediate subpart. (Again, we adopt the convention that we are only interested in smaller, simpler subparts.) The "parts-explosion" problem is to determine, for an arbitrary pair of parts x and y, how many y's appear in x. For example, if a bicycle has two wheels, and each wheel has forty-seven spokes, then we would like to infer that a bicycle has ninety-four spokes. We can solve the parts-explosion problem using the following program.

$$\begin{aligned} &in(X,Y,null,N) \leftarrow part(X,Y,N) \\ &in(X,Y,Z,N) \leftarrow part(X,Z,P), contains(Z,Y,M), N = P * M \\ &contains(X,Y,N) \leftarrow N = \textstyle\sum P : in(X,Y,Z,P) \end{aligned}$$

(The sum in the third rule is grouped by X and Y; for each X and Y we sum all corresponding P.) The sum operation here is not stratified. *contains* depends on itself through aggregation, via the predicate *in*. However, assuming *part* is acyclic in its first two arguments, the summation operates on successively lower arguments (i.e., smaller subparts), and so there is no looping through summation. This is the aggregate analog of modular stratification. □

2.3 Monotonicity Constraints

Looking at the examples of Section 2.2, it seems that in each of the examples there is a predicate that should be restricted to an acyclic relation for semantic reasons. Thus, for example, asking whether the program of Example 2.2 has a two-valued model for all possible values of EDB predicates is clearly the wrong question. The *part* relation represents a part hierarchy, and therefore can only be an acyclic relation. Thus we would like to phrase the question as whether a program has a two-valued well-founded model for all EDB relations that satisfy some acyclicity constraints. In

this section we present the notion of *monotonicity constraints* from [BS89] in order to be able to phrase such constraints.

We shall also look at the problem of *inferring* constraints on IDB predicates given constraints on the EDB predicates, extending some results from [BS89].

Definition 2.5: (Monotonicity Constraint) A *monotonicity constraint* is a statement of the form

$$p : i \prec_{mc} j$$

where p is a predicate name, and i and j are either column positions of p, or constants in the language (i.e., elements of U). The semantics of this construct is that for some partial order, column i (or the constant i) is less than column j (or the constant j) in each tuple of p. If both i and j are constants, then the semantics is that p is empty if the constraint $i \prec_{mc} j$ is violated. An *equality constraint* is a statement of the form

$$p : i =_{ec} j$$

where p is a predicate name, and i and j are either column positions of p, or constants in the language. A *disjunctive constraint* is a disjunction of conjunctions of monotonicity and equality constraints on a single predicate. □

Note that $\prec_{mc}$ does not represent a fixed partial order. We shall fix a partial order $<$ separately, and require predicates to satisfy some disjunctive constraints when $\prec_{mc}$ is replaced by $<$. (In Section 5 we discuss allowing a partial order that is not fixed in advance.) If $<$ is a fixed partial order, then we let $C_<$ represent a version of C in which $\prec_{mc}$ is replaced by $<$, and $=_{ec}$ is replaced by $=$.

We shall assume in this paper that all partial orders are *antireflexive* and *well-ordered*, so that $c \not\prec c$ for any constant c, and so that there are no infinite decreasing chains $\cdots \prec c_n \prec c_{n-1} \prec \cdots \prec c_1$. We shall use the term "constraint" to mean a disjunctive constraint, unless otherwise noted.

Brodsky and Sagiv give a set of axioms and inference rules for inferring monotonicity and equality constraints from a given set S of monotonicity and equality constraints, and show that the axiom system is sound and complete for consistent sets S. They also provide a sound and complete algorithm for inferring all disjunctive constraints that hold for an IDB predicate in a datalog program P given disjunctive constraints for the EDB predicates of P. Finally, they demonstrate that the problem of determining whether a monotonicity constraint on an IDB predicate is implied by a set of monotonicity constraints on the EDB predicates is complete for exponential time (although polynomial time if the arity of predicates is bounded).

Our context is more general than that of [BS89] since we allow negative subgoals.[1] While Brodsky and Sagiv's method is sound and complete for datalog programs, it may fail to detect monotonicity constraints for programs like

$$p(X, Y) \leftarrow q(X, Y), \neg q(X, Y).$$

Even though there are no constraints on relation q, the constraint $p : 1 \prec_{mc} 2$ is trivially satisfied because p must be empty. While one might imagine that inference rules could be added to detect rules of the form above, one can show that the implication problem for monotonicity constraints in programs with negation is undecidable. (This result will be presented separately.)

Nevertheless, Brodsky and Sagiv's algorithm is still sound if we apply it to the *envelope* of a program P. By removing all negative subgoals we can only enlarge the well-founded model (making atoms that were previously false or undefined true).

[1] Negative subgoals alone do not imply any monotonicity or equality constraints, since the complement of an antireflexive partial order is not, in general, an antireflexive partial order.

Theorem 2.2: Let P be a program and let P' be its envelope. Then the least model of P' contains all atoms that are either true or undefined in the well-founded model of P.
Proof: (This proof assumes some familiarity with the notation of [VGRS91].) Since P' is negation-free, its well-founded model is two-valued and equal to its least model by results of [VGRS91]. The negation-freeness of P' also means that the set of false atoms in the well-founded model for P' is $U_{P'}(\emptyset)$, where $U_Q(I)$ is the greatest unfounded set of program Q with respect to interpretation I.

Let M_P denote the well-founded model of P, and let $neg(I)$ denote the atoms that are false in I. We show that $neg(M_P) \supseteq U_{P'}(\emptyset)$.

$neg(M_P)$	$= U_P(M_P)$	Since M_P is the fixpoint.
	$\supseteq U_P(\emptyset)$	By the monotonicity of U_P.
	$\supseteq U_{P'}(\emptyset)$	Since there are fewer subgoals to satisfy in P', and hence fewer possible witnesses of unusability.

∎

Corollary 2.3: All constraints that hold for the envelope of P hold for P. ∎

Example 2.4: Let P be the program

$$p(X,Y) \leftarrow e(X,Y)$$
$$p(X,Y) \leftarrow p(X,Z), f(Z,Y), \neg p(Z,Y)$$

and let E be the constraint set $\{e : 2 \prec_{mc} 1, f : 2 \prec_{mc} 1\}$ on the EDB predicates. The envelope of P is

$$p(X,Y) \leftarrow e(X,Y)$$
$$p(X,Y) \leftarrow p(X,Z), f(Z,Y)$$

from which $p : 2 \prec_{mc} 1$ is derivable using the techniques of [BS89]. Thus, $p : 2 \prec_{mc} 1$ also holds for P. □

Definition 2.6: Let P be a program, E a constraint set on the EDB predicates of P, and I a constraint set on the IDB predicates of P. Let $<$ be a fixed antireflexive well-ordered partial order. We say $C = I \cup E$ is *sound* for $(P, <)$ if

- The EDB of P satisfies $E_<$, and
- For every atom A violating a constraint in $C_<$, A is false in the well-founded model for P.

□

Using Corollary 2.3, we can use Brodsky and Sagiv's inference method to derive a sound constraint set for any program. We just need to check that the EDB satisfies the constraints.

One might argue that checking that a constraint set is sound involves checking the data, not just the schema. This observation is correct; however, this EDB check would need to be performed by the database integrity subsystem anyway to ensure that the database is consistent, and so does not necessarily represent an additional burden. Further, such integrity constraint checks could be performed *incrementally*, and so have (incremental) cost significantly smaller than proportional to the size of the database.

3 Constraint Stratification: A Syntactic Condition

In this section we provide a syntactic stratification condition that is general enough to include all of the examples from Section 2.2 originally from [Ros90], while also including some programs that are not modularly stratified.

Definition 3.1: Let $<$ be a fixed antisymmetric partial order with no infinite descending chains. Let P be a program, and let C be a sound constraint set for $(P, <)$.

Let $inst(P, C, <)$ denote the instantiation of program P after all rules with a positive subgoal not satisfying $C_<$ are deleted. We say that P is *constraint stratified for* $(C, <)$ if and only if $inst(P, C, <)$ is locally stratified. □

Example 3.1: Consider the program P of Example 2.1 given by

$$w(X) \leftarrow m(X, Y), \neg w(Y)$$
$$m(a, b)$$
$$m(b, c)$$

Let $C = \{m : 2 \prec_{mc} 1\}$ be the constraint set, and let $<$ be given by $d < c < b < a$. $inst(P, C, <)$ is the program

$$w(a) \leftarrow m(a, b), \neg w(b)$$
$$w(a) \leftarrow m(a, c), \neg w(c)$$
$$w(a) \leftarrow m(a, d), \neg w(d)$$
$$w(b) \leftarrow m(b, c), \neg w(c)$$
$$w(b) \leftarrow m(b, d), \neg w(d)$$
$$w(c) \leftarrow m(c, d), \neg w(d)$$
$$m(a, b)$$
$$m(b, c)$$

which is locally stratified. Hence P is constraint stratified for $(C, <)$. □

Example 3.2: Consider the following program P from Example 2.2 with the constraint set $C = \{part : 2 \prec_{mc} 1\}$.

$$working(X) \leftarrow tested(X)$$
$$working(X) \leftarrow part(X, Y), \neg has\_suspect\_part(X)$$
$$has\_suspect\_part(X) \leftarrow part(X, Y), \neg working(Y)$$
$$part(a, b)$$
$$part(a, c)$$

Let the ordering $<$ be defined by $b < a$, $c < a$, and $d < a$ with b, c and d incomparable. (Assume for simplicity that these three constants are the only elements of $\mathcal{U}$.) $inst(P, C, <)$ is the program

$$working(a) \leftarrow tested(a)$$
$$working(b) \leftarrow tested(b)$$
$$working(c) \leftarrow tested(c)$$
$$working(d) \leftarrow tested(d)$$
$$working(a) \leftarrow part(a, b), \neg has\_suspect\_part(a)$$
$$working(a) \leftarrow part(a, c), \neg has\_suspect\_part(c)$$
$$working(a) \leftarrow part(a, d), \neg has\_suspect\_part(d)$$
$$has\_suspect\_part(a) \leftarrow part(a, b), \neg working(b)$$
$$has\_suspect\_part(a) \leftarrow part(a, c), \neg working(c)$$
$$has\_suspect\_part(a) \leftarrow part(a, d), \neg working(d)$$
$$part(a, b)$$
$$part(a, c)$$

which is locally stratified. Hence P is constraint stratified for $(C, <)$. □

Example 3.3: One can easily generalize the notions above to aggregation if one thinks of recursion through aggregation in a fashion similar to recursion through negation. Consider the program

$$
\begin{array}{l}
in(X, Y, null, N) \leftarrow part(X, Y, N) \\
in(X, Y, Z, N) \leftarrow part(X, Z, P), contains(Z, Y, M), N = P * M \\
contains(X, Y, N) \leftarrow N = \sum P : in(X, Y, Z, P) \\
part(a, b, 4) \\
part(a, c, 1) \\
part(b, c, 7)
\end{array}
$$

from Example 2.3. For simplicity, assume that the integers are all members of $\mathcal{U}$ with standard semantics for multiplication and sum. Let C be the constraint set $\{part : 2 \prec_{mc} 1, in : 2 \prec_{mc} 1, contains : 2 \prec_{mc} 1\}$, and let $<$ be defined by $c < b < a$ with all other elements incomparable. $inst(P, C, <)$ contains all instances[2] of the following program with respect to $\mathcal{U}$.

$$
\begin{array}{ll}
in(b, c, null, N) \leftarrow part(b, c, N) & in(a, b, null, N) \leftarrow part(a, b, N) \\
in(a, c, null, N) \leftarrow part(a, c, N) & \\
in(a, c, b, N) \leftarrow part(a, b, P), contains(b, c, M), N = P * M & \\
contains(b, c, N) \leftarrow N = \sum P : in(b, c, Z, P) & part(a, b, 4) \\
contains(a, c, N) \leftarrow N = \sum P : in(a, c, Z, P) & part(a, c, 1) \\
contains(a, b, N) \leftarrow N = \sum P : in(a, b, Z, P) & part(b, c, 7)
\end{array}
$$

Since $inst(P, C, <)$ is locally stratified with respect to aggregation, P is constraint stratified for $(C, <)$. □

One may wonder how one can derive that C is a sound constraint set in Example 3.3 above. That C is sound follows from the semantics of grouping variables in aggregation. One can argue that the derived constraints for the modified aggregation-free program

$$
\begin{array}{l}
in'(X, Y) \leftarrow part(X, Y, P) \\
in'(X, Y) \leftarrow part(X, Z, P), contains'(Z, Y) \\
contains'(X, Y) \leftarrow in'(X, Y)
\end{array}
$$

also hold (after translating the column positions) in the original program. The set $C = \{part : 2 \prec_{mc} 1, in' : 2 \prec_{mc} 1, contains' : 2 \prec_{mc} 1\}$ is derivable from the modified program using Brodsky and Sagiv's method.

This technique can be applied in general. One can form the aggregate-analog of the program envelope by projecting out all non-grouping variables from aggregate subgoals, and dropping the aggregate function. Attributes of other predicates that depend on the aggregated value may also have to be projected out, as illustrated above.

So far, every constraint stratified program we have seen has been modularly stratified. The following example is a program that is constraint stratified but not modularly stratified.

Example 3.4: Let P be the program

$$
\begin{array}{l}
p(X, Y) \leftarrow e(X, Y) \\
p(X, Y) \leftarrow p(X, Z), f(Z, Y), \neg p(Z, Y) \\
e(a, b) \\
f(b, c)
\end{array}
$$

[2]Note that one does not instantiate the non-grouping variables in an aggregate subgoal.

based on that of Example 2.4. Let C be the constraint set $\{e : 2 \prec_{mc} 1, f : 2 \prec_{mc} 1, p : 2 \prec_{mc} 1\}$. Note that $p : 2 \prec_{mc} 1$ is derivable for P from $\{e : 2 \prec_{mc} 1, f : 2 \prec_{mc} 1\}$ as discussed in Example 2.4. Let $<$ be defined by $c < b < a$. $inst(P, C, <)$ is

$$\begin{array}{l} p(a,b) \leftarrow e(a,b) \\ p(a,c) \leftarrow e(a,c) \\ p(b,c) \leftarrow e(b,c) \\ p(a,c) \leftarrow p(a,b), f(b,c), \neg p(b,c) \\ e(a,b) \\ f(b,c) \end{array}$$

Since $inst(P, C, <)$ is locally stratified, P is constraint stratified for $(C, <)$.

The program is not modularly stratified, since the following rule instance is in the reduction of p's component:

$$p(b,c) \leftarrow p(b,b), \neg p(b,c).$$

The rule instance above prevents the reduction from being locally stratified. Since the reduction only looks at lower-component predicates, there is no way to notice that $p(b,b)$ will never be satisfied. □

There are modularly stratified programs that are not constraint stratified.

Example 3.5: Consider the one-rule program

$$p \leftarrow q, \neg q, \neg p.$$

This program is modularly stratified: Since there are no rules for q, q is inferred false and the given rule does not appear in the reduction of p's component, thus allowing p to be inferred false. On the other hand, this program is not constraint stratified since monotonicity constraints "don't help" for predicates without arguments, and the rule above is not locally stratified. □

Theorem 3.1: Let C be a sound set of constraints on a program P, and let $<$ be a partial order. If P is constraint stratified with respect to $(C, <)$, then P has a two valued well-founded model whenever the EDB satisfies E.
Proof: By the soundness of the constraints, any instantiated rule with an atom violating C is false in the well-founded model. Hence that rule will not contribute to the well-founded model, and deleting it will give a program that is equivalent under the well-founded semantics. The program we eventually reach after deleting such rules is locally stratified, by the definition of constraint stratification, and hence has a two-valued well-founded model. ▮

Theorem 3.2: Every locally stratified program is constraint stratified with respect to any set of constraints.
Proof: Since this fact holds for the empty set of constraints. ▮

4 Optimization

The fundamental optimization technique for deductive databases is magic sets [BR91, BMSU86]. Magic sets speeds up bottom-up evaluation by passing binding information from the query into the rule evaluation so that only information relevant to the query is accessed.

Various authors have looked at the problem of extending the magic sets techniques to larger classes of programs with negation, be it stratified, modularly stratified, or general negation [Ros90, RSS92, Mor93, KSS92]. In general, the larger the class of programs allowed, the fewer options

there are for optimization, and so we look for the most specific optimization technique that applies. As an obvious example, one would not use any of the techniques for magic set with negation on a program without negation: it would be preferable to use the version defined for negation-free programs. As another example, techniques for evaluating programs with general recursion through negation have to deal with an additional truth value, undefined, that can represent a significant overhead for query evaluation.

Thus we seek an efficient optimization strategy for constraint stratified programs that is potentially better than those that apply to larger classes of programs. One doesn't have to look very far. It turns out that we can use techniques developed for modularly stratified programs [Ros90] if the program satisfies a slightly stronger condition about the way it passes its bindings from left to right in each rule.

Definition 4.1: Let $<$ be a fixed antisymmetric partial order with no infinite descending chains. Let P be a program, and let C be a sound constraint set for $(P, <)$.

Let *pref*$(P, C, <)$ denote the instantiation of *all prefixes of* program P after all rule prefixes with a positive subgoal not satisfying $C_<$ are deleted. We say that P is *constraint stratified from left to right for* $(C, <)$ if and only if *pref*$(P, C, <)$ is locally stratified. □

The definition of constraint stratification from left to right is analogous to the notion of modular stratification from left to right [Ros90]. The idea is to make sure that no recursive loops occur through the initial subgoals of a rule, even if the later subgoals would make the loop unsatisfiable.

Lemma 4.1: Every constraint stratified program can have its rules' subgoals reordered so that it is constraint stratified from left to right.
Proof: Call a subgoal "current" if it has a predicate from the same component as the head atom of the rule. For each rule place subgoals with lower-component predicates to the left of current subgoals. For negative current subgoals, move them to the right of all positive current subgoals. Since constraints come only from positive subgoals, all such constraints are applied in any instance of a rule prefix containing a negative current subgoal. Hence the constraint stratification of the original program implies the constraint stratification from left to right of the transformed program. ∎

While space restrictions prevent a full description of the magic sets techniques here, one can show that the techniques of [Ros90] (and, we conjecture, the techniques of [RSS92]) are correct for function-free programs that are constraint stratified from left to right. This result does not follow from results in [Ros90] since there exist constraint stratified programs that are not modularly stratified.

5 Conclusions

We have presented a syntactic condition for a deductive database to have a two-valued well-founded model. Previous conditions were either too weak to naturally express some programs, or were semantic in nature, depending on the values of certain relations in the database.

Our condition, called constraint stratification, is shown to be general enough to express all of the examples from [Ros90] of interesting modularly stratified programs. In addition, we demonstrate that there are some constraint stratified programs that are not modularly stratified. We can also show that magic sets optimization techniques developed for modularly stratified programs [Ros90] also apply to programs that are constraint stratified.

There are several directions for future work. The first problem is that of having to check all instances of all rules in order to determine constraint stratification. For programs with function

symbols the instantiated program is infinite. Even for function-free programs, checking all rule instances is likely to be a prohibitively expensive task. In a forthcoming paper [Ros94], we show that checking all rule instances is not necessary in general for function-free programs. One can manage just by looking at the original rules themselves in addition to the constraints.

The second problem is that this paper assumes that the partial order $<$ is given in advance. For some domains, such as the integers, this may be reasonable. For other domains, such as a part-subpart hierarchy, the order may depend on the database. For example, we know *engine* $<$ *torque-generator* only when the relation representing the hierarchy expresses that an engine is a direct or indirect subpart of a torque-generator. It is conceivable that in a different hierarchy, a torque-generator is a subpart of an engine.

In the same forthcoming paper [Ros94], we address this issue by quantifying over *all* partial orders that are consistent with the constraints, and requiring constraint stratification with respect to every such partial order.

Another direction for future work is on defining other sound (but still syntactic) constraint inference methods. Brodsky and Sagiv's method can potentially be extended to programs with negation and programs with function symbols to derive more constraints. Unfortunately, completeness is impossible since the monotonicity constraint implication problem for datalog programs with negation is undecidable [Ros94].

References

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, 1986.

[BR91] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–300, 1991. Preliminary version appeared in the 6th ACM Symposium on Principles of Database Systems, 1987.

[BS89] A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in Datalog programs. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, 1989.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, 1988.

[Kol91] P. G. Kolaitis. The expressive power of stratified programs. *Information and Computation*, 90:50–66, 1991.

[KSS92] D. Kemp, P. Stuckey, and D. Srivastava. Query restricted bottom-up evaluation of normal logic programs. In *Proc. Joint International Conference and Symposium on Logic Programming*, pages 288–302, 1992.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, 2nd edition, 1987.

[Mor93] S. Morishita. An alternating fixpoint tailored to magic programs. In *Proceedings of the Twelfth ACM Conference on Principles of Database Systems*, 1993. Preliminary version appeared in the 1992 Proceedings of the Workshop on Deductive Databases, Joint International Conference and Symposium on Logic Programming.

[PP90] H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990. Preliminary version appeared in Proc. Fifth International Conference and Symposium on Logic Programming, 1988.

[Prz88] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216, Los Altos, CA, 1988. Morgan Kaufmann.

[Ros90] K. A. Ross. Modular stratification and magic sets for Datalog programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, 1990. Full version to appear in J.ACM.

[Ros94] K. A. Ross. Stratification conditions using constraints. Submitted for publication, March 1994.

[RSS92] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Proc. Joint International Conference and Symposium on Logic Programming*, pages 273–287, 1992.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge Base Systems.* Computer Science Press, Rockville, MD, 1989. (Two volumes).

[VGRS91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.