A Syntactic Stratification Condition Using Constraints

Kenneth A. Ross
Department of Computer Science
Columbia University
New York, NY 10027
kar@cs.columbia.edu

Abstract

Stratification conditions for logic programs aim to ensure a two-valued semantics by restricting the class of allowable programs. Previous stratification conditions suffer from one of two problems. Some (such as modular stratification and weak stratification) are semantic, and cannot be recognized without examining the facts in addition to the rules of the program. Others (such as stratification and local stratification) are syntactic, but do not allow a number of useful examples. A nonsemantic version of modular stratification, i.e., whether a program is modularly stratified for all extensional databases, is shown to be undecidable. We propose a condition that generalizes local stratification, that ensures a two-valued well-founded model, and that can be syntactically determined from the rules and some constraints on the facts in the program. We call this condition "Universal Constraint Stratification." While not every modularly stratified program is universally constraint stratified, all of the well-known practical examples of modularly stratified programs are universally constraint stratified under appropriate natural constraints. In addition, there exist universally constraint stratified programs that are not modularly stratified.

Keywords: Negation, stratification, semantics, logic programming, constraints, deductive databases.

1 Introduction

Much recent work has concerned defining the semantics of negation in deductive databases. The "perfect model semantics" [5] 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 [9] and the stable model semantics [2] 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 [6] 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 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 [6]. 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, which is a syntactically recognizable condition.

One might try to make modular stratification nonsemantic by asking whether a set of rules is modularly stratified *for all possible extensional databases.* We prove that it is undecidable in general to determine whether a set of rules is modularly stratified for all possible extensional databases. Thus this version of modular stratification cannot be detected in general, and we need to look instead for a syntactically recognizable condition.

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 extensional database (EDB) predicates. The constraints we allow are monotonicity constraints [1]. 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 intensional database (IDB) predicates using a sound inference mechanism.

- Use these constraints to analyze all recursive loops through negation. If every recursive loop through negation needs an unsatisfiable constraint to hold in order for the loop to go through, then we can conclude that there is no "real" recursion through negation. Programs with this property are called "universally constraint stratified."

It is not obvious that this is a syntactic condition, i.e., that it can be effectively tested and that it depends on just the IDB and schema-level information. In the

outline above, it is conceivable that we may need to check infinitely many recursive loops through negation. We demonstrate that universal constraint stratification can be checked algorithmically.

Not every modularly stratified program is universally constraint stratified. However, we find, perhaps surprisingly, that all of the common examples of modularly stratified programs, including all of the examples from [6], 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, and that constraint stratification generalizes local stratification.

2 Terminology

We consider normal logic programs without function symbols [4], also known as "Datalog" programs with negation, and follow standard logic-programming conventions.

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 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 an infinite universe $\mathcal{U}$ is given. $\mathcal{U}$ should contain all constant symbols that can appear in all possible programs and EDB relations. In particular, $\mathcal{U}$ will include the Herbrand universe of any program/EDB pair. $\mathcal{U}$ will function as the domain under consideration, with terms interpreted freely. Even though $\mathcal{U}$ is infinite, only finitely many elements of $\mathcal{U}$ can be mentioned in any program. 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.

The mutually recursive components (also called strongly connected components) of a program have a natural relation associated with them: $F_1 \sqsubset F_2$ if some predicate belonging to F_1 is called by a predicate in 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 $\prec$, the transitive closure of $\sqsubset$, as the *dependency relation* between components. $\prec$ 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 $F' \prec F$.

Definition 2.1: The *envelope* of a program P is P with all negative subgoals removed. □

3 Modular Stratification

We now present the concept of modular stratification, originally defined in [6].

Definition 3.1: (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*. $\Box$

Definition 3.2: (Modular Stratification) Let $\prec$ be the dependency relation between components. We say that the program P is *modularly stratified* if, for every component F of P, (a) There is a total well-founded model M for the union of all components $F' \prec F$, and (b) The reduction of F modulo M is locally stratified. $\Box$

Theorem 3.1: ([6]) Every modularly stratified program has a total well-founded model that is its unique stable model. $\blacksquare$

3.1 Examples

We present a number of examples from [6] of modularly stratified programs. Note that none of these examples is locally stratified.

Example 3.1: 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 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. $\Box$

Example 3.2: 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:

$$\begin{array}{l} working(X) \leftarrow tested(X) \\ working(X) \leftarrow part(X,Y), \neg has_suspect_part(X) \\ has_suspect_part(X) \leftarrow part(X,Y), \neg working(Y) \end{array}$$

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. $\Box$

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

Example 3.3: 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. We can solve the parts-explosion problem using the following program.

$$in(X, Y, null, N) \leftarrow part(X, Y, N)$$
$$in(X, Y, Z, N) \leftarrow part(X, Z, P), contains(Z, Y, M), N = P * M$$
$$contains(X, Y, N) \leftarrow N = \sum P : in(X, Y, Z, P)$$

(The sum in the third rule is grouped by X and Y; for each X and Y we sum all corresponding P.) Assuming *part* is acyclic in its first two arguments, the program is modularly stratified with respect to aggregation. □

3.2 Modular Stratification is Undecidable

While modular stratification is a semantic condition, one may hope to achieve a nonsemantic condition by asking "Is program P modularly stratified for all EDBs?"

The main result of this section is that this version of modular stratification is an undecidable property of programs. As a result, one cannot, in general, recognize when a datalog program is modularly stratified for all EDBs. To establish this result we need a preliminary lemma, which uses a construction of Shmueli [8].

Lemma 3.2: It is recursively unsolvable to determine, for an arbitrary Datalog program whether a given relation is transitively closed for all possible assignments of relations to EDB predicates.

Proof: (Sketch) The first step is to show that testing whether a context free language L (not including the empty string[1]) has the property that $LL \subseteq L$, is undecidable. This proof uses standard techniques from [3].

The next step is to perform a translation from context-free grammars to datalog programs using Shmueli's construction. Containment of the corresponding context-free languages can then be expressed as the containment of a certain binary predicate g in the transformed program. Suppose that g is the predicate corresponding to the grammar for L, and that g' is the predicate corresponding to a derived grammar for LL. Then the structure of the transformation is such that g' is the composition of g with itself, and that $LL \subseteq L$ if and only if g is transitively closed. ∎

Theorem 3.3: It is recursively unsolvable to determine whether a datalog program is modularly stratified for all EDBs.

Proof: Consider a program P containing the rule

$$p(X) \leftarrow t(X, Y), t(Y, Z), \neg t(X, Z), \neg p(X)$$

together with some Datalog rules defining the IDB predicate t, but not mentioning p. Then P is modularly stratified if and only if t is transitively closed. Hence determining whether a program is modularly stratified for all values of EDB predicates is undecidable, by Lemma 3.2. ∎

Given that modular stratification for all EDBs is an undecidable property of programs, we need to find a syntactically recognizable condition that includes most of the useful modularly stratified programs discussed in the literature.

[1]We make the technical restriction that our context-free languages do not contain the empty string for compatibility with Shmueli's construction.

4 Monotonicity Constraints

Looking at the examples of Section 3.1, it seems that in each of the examples there is a predicate that should be restricted to an acyclic relation for semantic reasons. Thus, asking whether the program of Example 3.2 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. 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 [1] 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.

Definition 4.1: (Monotonicity Constraint) Let F be a logical formula and let S be the set of variables occurring in F. A *monotonicity constraint* is a statement of the form $F : X \prec_{mc} Y$, where X and Y are either members of S, or constants in the language. An *equality constraint* is a statement of the form $F : X =_{ec} Y$. A *disjunctive constraint* is a disjunction of conjunctions of monotonicity and equality constraints on a single formula F. If $F : C$ is a disjunctive constraint, then we refer to C as the *condition* of the constraint, and refer to F as the *formula* of the constraint. The empty disjunction is a constraint that is never satisfied. □

The intuition behind a monotonicity constraint $F : X \prec_{mc} Y$ is that for some partial order, argument X (or the constant X) is less than argument Y (or the constant Y) in each tuple of variables satisfying F. If X and Y are both constants, and $X \prec_{mc} Y$ is violated, then there are no tuples satisfying F. Equality constraints represent identity in $\mathcal{U}$; $a =_{ec} a$ for every $a \in \mathcal{U}$ but $a \neq_{ec} b$ for every distinct pair of constants a and b in $\mathcal{U}$. We shall write $F : (C1 \wedge C2)$ rather than $(F : C1) \wedge (F : C2)$, and similarly for disjunction ($\vee$).

Note that $\prec_{mc}$ does not represent a fixed partial order; the partial order may depend on the database instance. If $<$ is a fixed partial order, and C is a disjunctive constraint, then we let $C_<$ represent a version of C in which $\prec_{mc}$ is replaced by $<$, and $=_{ec}$ is replaced by $=$. ("$=$" represents syntactic equality.)

We shall assume in this paper that all partial orders are *finite* and *antireflexive*, so that $c \not< c$ for any constant c. In the context of function-free programs, the finiteness of the partial order is reasonable given the finiteness of the database. We shall use the term "constraint" to mean a disjunctive constraint, unless otherwise noted.

We have chosen a slightly more general notation for monotonicity constraints than that used in [1]. In [1], the authors effectively restrict the formula F in the definition above to be a single atom with distinct variables as arguments. We shall need the more general notation later in the paper.

Lemma 4.1: ([1]) It can be algorithmically determined for two disjunctive constraints C and D, whether $C \models D$. ▮

Definition 4.2: Let C be a constraint with formula F, and let S be the set of variables appearing in F. Let $S' \subseteq S$. The *projection* C' of C onto S' is a constraint with formula F and with the following properties:

- $C \models C'$, and
- If D is another constraint on the variables in S', and if $C \models D$, then $C' \models D$.

In other words, the projection of a constraint is the strongest possible constraint on the smaller set of variables that follows from the original constraint. □

Lemma 4.2: ([1]) The projection of a constraint always exists, is unique up to equivalence, and can be effectively computed. ▮

Example 4.1: Let F be the predicate $p(X,Y,Z,W)$, and let C be the constraint

$$F : (a \prec_{mc} Z) \wedge (X \prec_{mc} Z) \wedge (Z \prec_{mc} Y) \wedge (Y \prec_{mc} b) \wedge (W =_{ec} c) \wedge (W \prec_{mc} b)$$

where a, b and c are constants. Then the projection C' onto $\{X, Y\}$ is given by

$$F : (a \prec_{mc} Y) \wedge (X \prec_{mc} Y) \wedge (Y \prec_{mc} b) \wedge (c \prec_{mc} b) \quad \Box$$

Definition 4.3: A substitution σ from a set of variables S to a range $\mathcal{V}$ is a map from elements of S to values in $\mathcal{V}$. A constraint C is *satisfiable* if there exists a substitution σ and a partial order $<$ such that $C_{<}\sigma$ is true; otherwise C is said to be unsatisfiable. □

Lemma 4.3: ([1]) A constraint C is unsatisfiable iff it is inconsistent. ▮

Our context is more general than that of [1] since we allow negative subgoals.[2] While Brodsky and Sagiv's method is sound and complete for datalog programs, it may fail to detect monotonicity constraints for programs like

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

Even though there are no restrictions on relation e, the constraint $p(X,Y) : X \prec_{mc} Y$ 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 inference problem for monotonicity constraints in programs with negation is undecidable using a construction similar to that used in Theorem 3.3.

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).

Lemma 4.4: ([7]) 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. ▮

Thus, any constraints that hold for the envelope of P certainly hold for P.

Example 4.2: Let P be the program

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

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

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

from which $p(X,Y) : Y \prec_{mc} X$ is derivable using the techniques of [1]. Thus, $p(X,Y) : Y \prec_{mc} X$ is true in P given E. □

Definition 4.4: Let P be a program, E a constraint set on the EDB predicates of P, and C a constraint set on the IDB predicates of P. We say that $C \cup E$ is *sound* for P if all constraints in C are consequences of E given P. □

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

4.1 Constraints on Rules

So far the formulas F in constraints have been atoms with distinct variables as arguments. We shall also be interested in allowing rules as values for F. Specifically, suppose r is a rule and S is the set of variables in r. Then we shall be interested in constraints of the form

$$r : Y \prec_{mc} Z \quad \text{and} \quad r : Y =_{ec} Z$$

where Y and Z are either constants or members of S.

We can infer a constraint on rule r from a constraint set D on the predicates appearing positively in the body of r. For example, given the constraints $p(X,Y) : Y \prec_{mc} X$, $q(X,Y) : Y \prec_{mc} b$, $t(X) : X =_{ec} a$, $s(X,Y,Z) : Y =_{ec} Z$ and the rule r with body $p(W,X), q(X,Y), t(Y), s(Y,a,V)$ we can infer the constraint

$$r : (X \prec_{mc} W) \wedge (Y \prec_{mc} b) \wedge (Y =_{ec} a) \wedge (V =_{ec} a).$$

There is a sound, complete, and effective way to perform this inference using techniques from [1]. We shall assume that we have performed this inference for every rule r.

We may also want to "go the other way" by projecting a constraint on a rule onto the arguments of some of its atoms.

Definition 4.5: Let r be a rule, and let D be a set of constraints on the predicates in r. We choose two occurrences of predicates in r: the head predicate and a body predicate. Let p_h be the head predicate occurrence, and q_b be the body predicate occurrence. (If there is more than one occurrence of q in the body then we need to specify which one.) If q takes m arguments, then we invent m new variables $q_b : 1, \ldots, q_b : m$. Similarly, if p takes j arguments, then we invent j new variables $p_h : 1, \ldots, p_h : j$.

Let F be the formula $foo(q_b : 1, \ldots, q_b : m, p_h : 1, \ldots, p_h : j)$ where *foo* is some new, arbitrary predicate symbol. Let C be the constraint of r given D. Append (i.e., conjoin) to the condition of C the conjunction

$$(q_b : 1 = q^1) \wedge \cdots \wedge (q_b : m = q^m) \wedge (p_h : 1 = p^1) \wedge \cdots \wedge (p_h : j = p^j)$$

where q^i is the ith argument of q_b in r and p^i is the ith argument of p_h in r.

After transforming the above constraint to disjunctive normal form, we project it onto the variables $\{q_b : 1, \ldots, q_b : m, p_h : 1, \ldots, p_h : j\}$ to get a new constraint C'. Finally, the formula in C' is replaced by F. We call C' the *projection* of C onto (p_h, q_b), denoted $\pi_D(r, p_h, q_b)$. □

Example 4.3: Let r be the rule

$$t(W,Y) \leftarrow p(W,X), q(X,Y), s(Y)$$

and suppose we have the constraint C on rule r given by

$$r : (X \prec_{mc} W) \wedge (Y \prec_{mc} b) \wedge (Y =_{ec} a)$$

as above. We project r onto (t_h, s_b). After appending the conjunction to C we have

$$r : (X \prec_{mc} W) \wedge (Y \prec_{mc} b) \wedge (Y =_{ec} a) \wedge (t_h : 1 =_{ec} W) \wedge (t_h : 2 =_{ec} Y) \wedge (s_b : 1 =_{ec} Y)$$

which is in disjunctive normal form. After the projection and after replacing the formula we get

$$foo(t_h : 1, t_h : 2, s_b : 1) : (t_h : 2 \prec_{mc} b) \wedge (t_h : 2 =_{ec} a) \wedge (t_h : 2 =_{ec} s_b : 1) \quad \square$$

5 Universal Constraint Stratification

In this section we define a stratification condition that is general enough to include all of the examples from Section 3.1 originally from [6], while also including some programs that are not modularly stratified. In a later section, we shall demonstrate that the condition is syntactic.

The intuition behind our stratification condition is that we shall start with a set of constraint formulas describing the EDB of a program. Given $e(X,Y) : Y \prec_{mc} X$ as a constraint, we know that e represents an acyclic relation. As discussed in Section 3.1, acyclicity is often a very natural restriction to place on a relation.

In general, we may also infer some predicate constraints for the IDB relations. For example, the constraint $p(X,Y) : Y \prec_{mc} X$ is a consequence of the program

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

and the constraint on e. Thus we shall have a set C of constraints describing the EDB and IDB predicates of the program.

From this point we shall try to demonstrate that a recursive loop through negation is impossible. This is done by showing that when a predicate p recurses through negation, *it is impossible according to the constraints that the arguments of p in the head are the same as the arguments of p in the body.* We must consider both direct recursion through negation, and recursion through negation via other mutually recursive predicates.

Definition 5.1: Let D be a set of constraints on the predicates of program P. We construct a graph, called the *predicate constraint graph of P with respect to D*, whose nodes are predicates from P. Arcs in the predicate constraint graph have two properties: a polarity, either positive or negative, and a constraint. There is an arc from q to p with positive polarity if

$$(r) : \quad p(\cdots) \leftarrow \cdots, q(\cdots), \cdots$$

is a rule in P and p and q are mutually recursive. There is an arc from q to p with negative polarity if

$$(r) : \quad p(\cdots) \leftarrow \cdots, \neg q(\cdots), \cdots$$

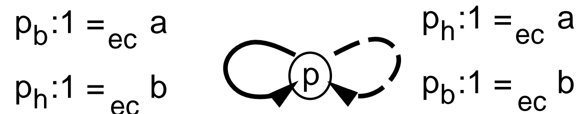
is a rule in P and p and q are mutually recursive. An arc corresponding to rule r is labeled with $\pi_D(r, p_h, q_b)$. □

Graphically, we shall illustrate a positive arc as a solid arrow, and a negative arc as a dashed arrow. The constraint of an arc will be shown as a label on the arc. We shall omit the conjunction symbol and the constraint's formula in the graphs when the meaning is clear.

Example 5.1: Consider the program P_1 given by

$$(r_1) \quad p(a) \leftarrow \neg p(b)$$
$$(r_2) \quad p(b) \leftarrow p(a)$$

with no additional constraints. The predicate constraint graph of this program is



Consider the program P_2 given by

$$\begin{array}{ll}(r_1) & p(a) \leftarrow e(a,b), \neg p(b) \\ (r_2) & p(b) \leftarrow e(b,a), p(a)\end{array}$$

with the monotonicity constraint $\{e(X,Y) : (X \prec_{mc} Y) \vee (X =_{ec} Y)\}$. Then the predicate constraint graph is

$p_b{:}1 =_{ec} a$ | | $p_b{:}1 =_{ec} a$ | | $p_h{:}1 =_{ec} a$ | | $p_h{:}1 =_{ec} a$
$p_h{:}1 =_{ec} b$ | **OR** | $p_h{:}1 =_{ec} b$ | (p) | $p_b{:}1 =_{ec} b$ | **OR** | $p_b{:}1 =_{ec} b$
$b <_{mc} a$ | | $a =_{ec} b$ | | $a <_{mc} b$ | | $a =_{ec} b$

□

5.1 Composing Constraints

In this section we show how to compose constraints on arcs of the predicate constraint graph. The notion of constraint composition will be used to define our stratification condition.

Definition 5.2: Let D be a set of constraints on predicates of a program P. Let $S_1 = \pi_D(r_1, p_h, q_b)$ and $S_2 = \pi_D(r_2, q_h, t_b)$ be constraints. Let j, m, k be the respective arities of p, q, t. Let C_1 and C_2 be the respective conditions of S_1 and S_2. The *sequential composition* of S_1 and S_2, denoted $\pi_D(r_1 \circ r_2, p_h, t_b)$, is defined as follows.[3]

First, if there is any name conflict between p_h and q_h, or between q_b and t_b, then we replace the symbol q with a new symbol q', of the same arity as q, consistently in S_1 and S_2. The condition of the sequential composition is the projection onto $\{p_h : 1, \cdots, p_h : j, t_b : 1, \cdots, t_b : k\}$ of the disjunctive normal form of

$$C_1 \wedge C_2 \wedge (q_h : 1 =_{ec} q_b : 1) \wedge \cdots \wedge (q_h : m =_{ec} q_b : m).$$

The sequential composition's formula is *foo*$(p_h : 1, \cdots, p_h : j, t_b : 1, \cdots, t_b : k)$. The sequential composition of a sequence of constraints is defined as the successive pairwise sequential composition of its elements. Sequential composition is associative (up to equivalence): thus the order of pairwise composition is unimportant. □

Definition 5.3: Let D be a set of constraints. Let $\pi_D(r_1, p_h^1, p_b^2)$, $\pi_D(r_2, p_h^2, p_b^3)$, $\ldots$, $\pi_D(r_n, p_h^n, p_b^1)$ be a sequence of constraints, where each p^i is a (not necessarily distinct) predicate symbol, and each r_i is a (not necessarily distinct) rule. The *cyclic composition* of this sequence is denoted by $\theta_D(r_1, \cdots, r_n)$ and is defined as follows. Let C be the condition of the sequential composition

$$\pi_D(r_1 \circ \cdots \circ r_n, p_h^1, p_b^1).$$

The condition of the cyclic composition is given by the projection onto the empty set of variables of the disjunctive normal form of the formula

$$C \wedge (p_h^1 : 1 =_{ec} p_b^1 : 1) \wedge \cdots \wedge (p_h^1 : k =_{ec} p_b^1 : k).$$

where the arity of p^1 is k. The formula of the cyclic composition is the variable-free formula *foo*. □

[3] Strictly speaking, our notation should specify the particular *occurrence* of q_b in r_1. We omit this information in order to make the notation simpler.

Example 5.2: Consider the programs P_1 and P_2 from Example 5.1. For P_1, the sequential composition $\pi_\emptyset(r_1 \circ r_2, p_h, p_b)$ is the projection of

$$(p_h : 1 =_{ec} b) \wedge (p'_b : 1 =_{ec} a) \wedge (p'_h : 1 =_{ec} a) \wedge (p_b : 1 =_{ec} b) \wedge (p'_b : 1 =_{ec} p'_h : 1)$$

onto (p_h, p_b). Note the renaming of the intermediate p to p'. The result is

$$(p_h : 1 =_{ec} b) \wedge (p_b : 1 =_{ec} b).$$

The cyclic composition is then the projection onto the empty set of variables of

$$(p_h : 1 =_{ec} b) \wedge (p_b : 1 =_{ec} b) \wedge (p_h : 1 = p_b : 1)$$

which can be represented as the trivially satisfiable constraint $b =_{ec} b$.

For P_2, let D be the constraint set $\{e(X, Y) : (X \prec_{mc} Y) \vee (X =_{ec} Y)\}$. One may verify that the cyclic composition $\theta_D(r_1, r_2)$ has condition

$$(b \prec_{mc} a \wedge a \prec_{mc} b) \vee (b \prec_{mc} a \wedge a =_{ec} b) \vee (a =_{ec} b \wedge a \prec_{mc} b) \vee (a =_{ec} b \wedge a =_{ec} b)$$

which is unsatisfiable since $\prec_{mc}$ represents a strict partial order, and since different constants cannot be equal under $=_{ec}$. □

5.2 Universal Constraint Stratification

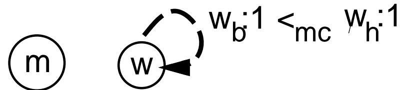
Definition 5.4: Let D be a sound set of constraints on the predicates in P, and let G be the predicate constraint graph of P with respect to D. We say that P is *universally constraint stratified with respect to* D if for every cycle in the predicate constraint graph that contains a negative arc, the cyclic composition of the constraints on the arcs in the cycle is unsatisfiable. □

Example 5.3: Consider once more the programs P_1 and P_2 from Example 5.1. As we saw in Example 5.2, P_1 has a cycle with a negative arc whose cyclic composition yields a satisfiable constraint. This cycle corresponds to the intuition that $p(a)$ depends negatively on itself.

On the other hand, no cycle containing a negative edge in P_2 is satisfiable. We leave it as an exercise for the reader to verify this claim. As a consequence, P_2 is universally constraint stratified with respect to the given constraints. The intuition here is that even though $p(a)$ apparently depends negatively on itself (through $p(b)$), in order to achieve this recursion through negation one would need both $e(a, b)$ and $e(b, a)$ to simultaneously be true. These two atoms cannot be simultaneously true given $e(X, Y) : (X \prec_{mc} Y) \vee (X =_{ec} Y)$. □

We now consider the examples from Section 3.1

Example 5.4: Given $C = \{m(X, Y) : (Y \prec_{mc} X)\}$ the program of Example 3.1 has predicate constraint graph

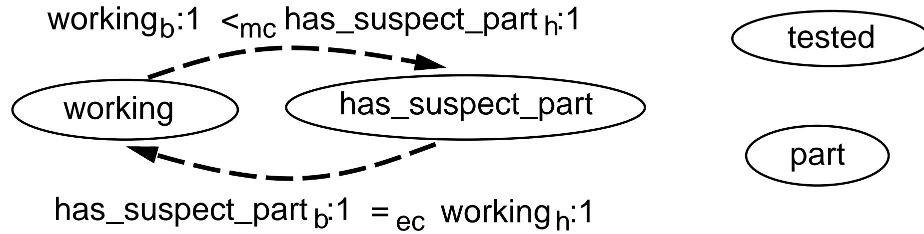


The program is universally constraint stratified with respect to C because all constraints of the form

$$(w'_b : 1 \prec_{mc} w_h : 1) \wedge (w'_b : 1 =_{ec} w'_h : 1) \wedge (w''_b : 1 \prec_{mc} w'_h : 1) \wedge$$
$$(w''_b : 1 =_{ec} w''_h : 1) \wedge \cdots \wedge (w_b : 1 \prec_{mc} w'^{\cdots\prime}_h : 1) \wedge (w_b : 1 =_{ec} w_h : 1)$$

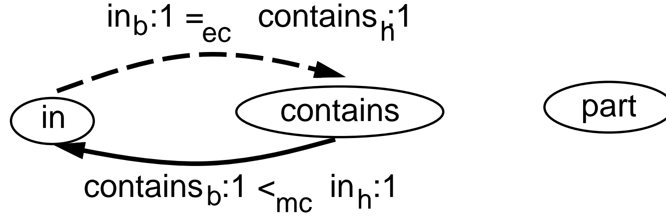
(corresponding to traversing the negative arc as many times as there are primed versions of the w predicate) are unsatisfiable.

Given the constraint set $C = \{part(X,Y) : (Y \prec_{mc} X)\}$, the program of Example 3.2 has predicate constraint graph



and is universally constraint stratified with respect to C since (as the reader may verify) the cyclic composition of the constraints on the cycles is unsatisfiable.

One can easily generalize the notions above to aggregation if one labels an edge as negative if it corresponds to recursion through either aggregation or negation. In that case, the predicate constraint graph for the program of Example 3.3 with respect to the constraint set $C = \{part(X,Y) : (Y \prec_{mc} X)\}$ is



(In the graph above we have left out some constraints on the arcs for simplicity of presentation. For example, in both constraints there should be a conjunct equating the second arguments of *contains* and *in*.) Again, the program is universally constraint stratified with respect to C. □

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

Example 5.5: 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 the constraint set $\{e(X,Y) : (Y \prec_{mc} X),\ f(X,Y) : (Y \prec_{mc} X),\ p(X,Y) : (Y \prec_{mc} X)\}$ be denoted by C. Note that $p(X,Y) : (Y \prec_{mc} X)$ is derivable for P from $\{e(X,Y) : (Y \prec_{mc} X),\ f(X,Y) : (Y \prec_{mc} X)\}$ as discussed in Example 4.2. Then the predicate constraint graph for P is (omitting some irrelevant parts of the constraints)

$p_h{:}1 =_{ec} p_b{:}1$ p $p_b{:}1 <_{mc} p_h{:}1$ (e) (f)

whose negative cycles have unsatisfiable cyclic compositions. The program is not modularly stratified, since it may be possible to have the following rule instance in its reduction:

$$p(a,b) \leftarrow p(a,a), \neg p(a,b)$$

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

An example of a program that is modularly stratified but not constraint stratified is the single-rule program $p \leftarrow q, \neg p$.

Theorem 5.1: Let C be a sound set of constraints on a program P, and let E be the constraints from C on EDB predicates. If P is constraint stratified with respect to C, and if there exists a partial order $<$ for which the EDB satisfies $E_<$, then P has a two valued well-founded model. ▮

While the converse of Theorem 5.1 is not necessarily true, we can show the following result.

Theorem 5.2: Let C be a sound set of constraints on a program P, and let E be the constraints from C on EDB predicates. If P is not constraint stratified with respect to C, then there exists a partial order $<$ and an EDB satisfying $E_<$ such that

- There is some set P' of instantiated rules from P whose positive body atoms satisfy $C_<$, and
- P' is not locally stratified. ▮

Theorem 5.2 implies that if a program is not constraint stratified then there is a potential loop through negation that cannot be ruled out on the basis of the constraints.

Theorem 5.3: Every (range-restricted, function-free) locally stratified program is universally constraint stratified with respect to the empty set of constraints. ▮

6 Universal Constraint Stratification is Syntactic

In this section we demonstrate that universal constraint stratification is syntactic. We say that a condition on logic programs is *syntactic* if (a) The condition can be determined algorithmically, i.e., it is a decidable property, and (b) The condition depends only upon the IDB and schema-level information about the EDB. By schema-level information we mean a fixed collection of information about the EDB that does not change as the EDB gets larger. In particular, the constraints we use in this paper fit this description.

Since the IDB and schema-level information is likely to be small compared to the size of the EDB, checking that a syntactic condition is satisfied is likely to be easier than a condition that needs to examine the EDB. Further, while the EDB

may change frequently over time, the IDB and schema-level information is likely to remain constant over relatively long periods of time, and hence a syntactic condition does not have to be re-checked often.

It is clear that universal constraint stratification satisfies the second condition for being syntactic, since it depends upon only the IDB and the constraints. We need to verify that it is algorithmically decidable.

There are two potential issues that need to be resolved in order to demonstrate that universal constraint stratification is decidable. First, we need to check that each of the steps used in transforming constraints can be performed algorithmically. These transformations include inferring constraints, performing projections, performing compositions, and testing for satisfiability. That each of these steps is algorithmic follows from Lemmas 4.2 and 4.3, and from the soundness, completeness, and decidability of the constraint inference problem as shown in [1].

The second issue is that universal constraint stratification requires that all cycles with a negative edge have constraints whose cyclic composition is unsatisfiable. Even for simple programs it is possible that there are infinitely many such cycles. Thus we need to demonstrate that we can determine the unsatisfiability of all cycles in finite time.

We can resolve the second issue by observing that for a given program P and a given constraint set D, there are finitely many nonequivalent constraints that use constants from P and D and any fixed set of variables. One can algorithmically determine whether two constraints are equivalent, by Lemma 4.1.

The cyclic composition θ is formed by taking a sequential composition π and adding some constraints. π may be replaced by any equivalent constraint and yield an equivalent cyclic composition θ. Thus we can try to find all sequential compositions, and then use these to form the cyclic compositions and test for unsatisfiability. We need check only finitely many sequential compositions by using the following procedure:

1. For each node in the predicate constraint graph, construct all simple cycles, and calculate the sequential composition on those cycles.

2. For each node in the predicate constraint graph, construct all cycles with exactly one sub-cycle, and calculate the sequential composition on those cycles. If at least one of these cycles yields a new sequential composition then continue. Otherwise, stop; all other cycles will have a sequential composition that will be equivalent to a previously derived one.

3. Repeat the previous step for 2 sub-cycles, 3 sub-cycles, and so on. Since there are only finitely many inequivalent constraints, we must eventually terminate.

Theorem 6.1: Universal constraint stratification is syntactic. ▮

7 Related Work

In [7] the present author defined the notion of "constraint stratification." That notion is more restricted than universal constraint stratification in two ways.

First, constraint stratification requires that a partial order $<$ be specified *in advance*. Thus, for constraint stratification, it is not possible to simply state that the relation *part* is an acyclic relation; one must know the partial order with respect to which *part* is acyclic. This is often an unreasonable restriction since we may want to assume that *part* is an acyclic relation representing a part hierarchy that *defines*

the partial order. We do not want to have to commit to a partial order that may need to be changed when the EDB changes.

In this paper we do not suffer from this problem because we define universal constraint stratification in terms of an arbitrary partial order. This extension leads to much of the technical complexity of this paper.

A second way that this paper improves upon [7] is that, unlike [7], one does not need to check all instances of a set of rules in order to determine universal constraint stratification. One applies constraint techniques to the uninstantiated rules.

As far as the author is aware, there is no other proposed stratification condition on datalog programs that simultaneously (a) Is syntactic, (b) Generalizes local stratification, (c) Ensures a two-valued well-founded model, and (d) Admits the examples from [6] (Section 3.1) of useful programs that are not locally stratified.

Acknowledgements

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

References

[1] 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.

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

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, MA, 1979.

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

[5] 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.

[6] 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.

[7] K. A. Ross. Constraint stratification. In *Proceedings of the ICLP Workshop on Deductive Databases*, pages 101–116, June 1994. Available as German GMD Society Publication Series number 231, ISBN 3-88457-231-8.

[8] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, 1987.

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