Structural Totality and Constraint Stratification

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

Abstract

In previous work we proposed 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 monotonicity constraints on the facts in the program. This condition was called "universal constraint stratification." In this paper we generalize the class of constraints from monotonicity constraints to an arbitrary constraint domain. We generalize universal constraint stratification to the more general case, and prove that the condition always ensures a two-valued well-founded model for finite databases. We also provide an alternative characterization of local stratification using a version of universal constraint stratification with equality constraints. We extend Papadimitriou and Yannakakis' notion of structural totality. Papadimitriou and Yannakakis proved that a program is structurally total if and only if it is stratified. We extend their notion of structural totality so that schema-level constraint information about the variables in the rules is considered part of the "structure." We prove that, for constraints expressed in certain "exact" constraint domains, a program satisfies our extended notion of structural totality if and only if it is universally constraint stratified.

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

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

1 Introduction

The well-founded semantics has been accepted as a natural semantics for deductive databases with negation or aggregation [VGRS91, Ros94a]. The well-founded semantics is a three-valued semantics. Literals may be *true*, *false* or *undefined*. 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. Additionally, programs with a two valued well-founded model have a unique stable model [GL88, VGRS91]. So we desire conditions on the program that ensure that the well-founded semantics is two-valued.

A stratified program is one in which, effectively, there is no predicate that depends negatively on itself. This simple, syntactic condition does ensure a two-valued well-founded model. However, stratification is not general enough to express a number of interesting and useful examples. Even when slightly generalized to "local stratification" [Prz88], we do not obtain sufficient expressive power.

The class of *modularly stratified* programs was proposed in [Ros94a]. 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. Other stratification conditions, such as weak stratification [PP90], dynamic stratification [Prz89], and effective stratification [BF91] are also semantic.

One might try to make modular stratification non-semantic by asking whether a set of rules is modularly stratified *for all possible extensional databases*. Unfortunately, this problem is undecidable [Ros94b]. 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 [Ros94b] a syntactic condition is proposed that is more general than local stratification. The programmer specifies some schema-level constraints on the extensional database (EDB) predicates. The constraints allowed in [Ros94b] are monotonicity constraints [BS89]. Monotonicity constraints specify that one argument of a predicate is less than another according to some partial order. The 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."

In this paper, we generalize this approach so that it can be applied for an arbitrary constraint domain, rather than only for monotonicity constraints. In order to do so, we may need to "approximate" certain steps in the process of determining universal constraint stratification since the individual steps may not be decidable in general. (We call a constraint domain *exact* if all of these intermediate steps are decidable and can be done exactly.) We show that a program has a two-valued well-founded model if it satisfies this more general definition of universal constraint stratification, which is also shown to be syntactic. Additionally, we find an alternative characterization of local stratification as a form of universal constraint stratification over a constraint domain with only equality constraints. This alternative characterization means that it is possible to determine whether a function-free program is locally stratified without instantiating the whole program.

We then address the issue of structural totality. Papadimitriou and Yannakakis defined the notion of structural totality to mean, informally, that a program has a two-valued well-founded model *for structural reasons* [PY92]. In other words, a program is structurally total if every program with the same "structure" has a two-valued well-founded model. Their notion of structure was the *skeleton* of the program, i.e., the propositional program formed by removing all arguments. The aim of choosing the skeleton as the "structure" was to avoid the "intricate pattern in which variables and constants repeat in the rules," since this pattern of variables and constants was presumably not part of the structure. One of the main results of [PY92] is that (under some mild conditions) a program is structurally total if and only if it is stratified.

In this paper we question the notion that the structure of a program is fully represented by the skeleton. In our view, there may be integrity constraints on the EDB predicates that restrict the pattern of variables and constants in the rules to be of a particularly ordered form. For example, a monotonicity constraint on a part-subpart hierarchy may ensure that when a predicate recurses it recurses *down the hierarchy*. We argue that such constraints (or constraints derived from them) really do belong to the "structure" of a program.

Thus we represent the "structure" of a program not as its skeleton, but using a more general notion of a *predicate constraint graph*. We shall define an extended notion of structural totality corresponding to the property that every program with the same predicate constraint graph has a two-valued well-founded model. The main result about structural totality is that for exact constraint domains, our notion of universal constraint stratification coincides with the extended notion of structural totality.

2 Preliminaries

We consider deductive databases without function symbols [Ull89], also known as "Datalog" programs with negation, and follow standard deductive database conventions. 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.

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.

Definition 2.1: The *skeleton* of a program P is P with all parentheses, variables, and constants omitted. Thus the skeleton of P is a propositional program. The *envelope* of a program P is P with all negative subgoals removed. Let P' be the skeleton of the envelope of P. Let p be a predicate in P (and hence in P'), and let F be the mutually recursive component to which p belongs in P'. We say p is *useless* in P, if F contains no rules whose body contains only lower-component predicates. A program is *useless-free* if it has no useless predicates. □

2.1 Motivating Examples

We present a number of examples of modularly stratified programs; the first two are from from [Ros94a]. Note that none of these examples is locally stratified.

Example 2.1: 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 express this in the following program:

$$\begin{array}{l} \mathit{working}(X) \leftarrow \mathit{tested}(X) \\ \mathit{working}(X) \leftarrow \mathit{part}(X,Y), \neg \mathit{has\_suspect\_part}(X) \\ \mathit{has\_suspect\_part}(X) \leftarrow \mathit{part}(X,Y), \neg \mathit{working}(Y) \end{array}$$

This program is modularly 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. □

Example 2.2: Let P be the program

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

together with a set of facts for the EDB predicates b and e. X, Y and Z are integers from some range $1,\ldots,n$, and $e(X,Y,Z)$ is true precisely when Y and Z are proper factors of X, i.e., when $YZ = X$ and both X and Y are greater than 1. $b(X)$ is true precisely when X is prime. One may verify that this program is modularly stratified, and that $p(X)$ is true (according to the well-founded semantics) when X is the product of an odd number of primes, and false otherwise. □

Example 2.3: Let P be the program

$$\begin{array}{l} \mathit{fib}(1) \\ \mathit{fib}(2) \\ \mathit{fib}(X) \leftarrow s(X,Y,Z), \mathit{fib}(Y), \mathit{fib}(Z), s(Z,Y,V), \\ \qquad\qquad\qquad\qquad\qquad\qquad \neg \mathit{between}(Y,Z) \\ \mathit{between}(Y,Z) \leftarrow b(Y,W,Z), \mathit{fib}(W) \end{array}$$

together with a set of facts for the EDB predicates s and b. The domain of the attributes of s and b is the integers from some range $1,\ldots,n$. $s(X,Y,Z)$ is true precisely when $Y+Z = X$. $b(X,Y,Z)$ is true precisely when $X < Y < Z$. (The subgoal $s(Z,Y,V)$ in the third rule is to ensure that $Y < Z$.) One may verify that this program is modularly stratified, and that $\mathit{fib}(X)$ is true (according to the well-founded semantics) when X is a Fibonacci number in the range $1,\ldots,n$. □

3 Constraint Domains

In this section we define the notion of a constraint domain.

Definition 3.1: A *constraint language* L on a domain D, is a first-order language whose terms are constants, variables and complex terms over D, having the following properties: (a) L includes equality (written $=$) of terms in D, interpreted as identity in D, and (b) L includes constraints that are universally true (tautologies) and constraints that are universally false (unsatisfiable constraints), and (c) L is closed under conjunction and variable renaming.

The pair (L, D) is called a *constraint domain*. An element of a constraint language is called a *constraint formula*. We assume a standard semantics of the *satisfaction* of constraint formulas and the *semantic implication* ($\models$) of constraint formulas for each standard domain.

The *solution* of a constraint formula C with free variables $X_1,\ldots,X_n$ is the set of n-tuples of values from D that satisfy C when substituted for $X_1,\ldots,X_n$ respectively. □

The restrictions on constraint languages in Definition 3.1 are standard [JM94]. Unlike [JM94], we do not require that the constraint language is closed under existential quantification.

A constraint language L on a domain D typically contains (a) A collection of basic constraints that can be asserted between elements of D, such as equality, inequality, etc.; (b) A set of operations by which constraints can be combined to form compound constraints, such as disjunction, conjunction and negation; (c) A set of quantifiers that can be applied to constraints, usually existential and universal quantification; and (d) A language in which terms from domain D may be constructed from other terms, such as addition and multiplication of numbers.

In what follows, we shall use the term "constraint" to mean "constraint formula" unless otherwise noted.

Definition 3.2: A constraint will be written as

$$C(X_1, \cdots, X_n)$$

where C is an expression belonging to the constraint language. A *predicate* constraint formula is a constraint formula of the form $p(X_1, \cdots, X_n) : C(X_1, \cdots, X_n)$ where p is a predicate of arity n. □

Predicate constraint formulas describe single relations. For example, the predicate constraint formula $e(X, Y) : X = Y$ would be satisfied by e if every tuple t in e has first component equal to the second component. Thus, in a predicate constraint the variables are implicitly *universally* quantified.

Example 3.1: (Equality Constraints) The most basic constraint language is one based on equality of members of the domain. In this language, the only basic constraint is equality. $a = b$ if and only if a and b are the same element. Compound operations can be formed by taking conjunctions or disjunctions of other constraints, with the standard semantics. □

Example 3.2: (Partial Order Constraints) Another constraint language is one based on equality and order of members of a domain D that has a fixed antisymmetric partial order $<$. In this language, the basic constraints are equality, and order $<$. $a < b$ if and only if a is less than b according to $<$. Compound operations can be formed by taking conjunctions or disjunctions of other constraints, with the standard semantics. □

Example 3.3: (Monotonicity Constraints) [BS89] A less standard constraint language is one based on equality and order of members of a domain D, where the order is based on *some* partial order rather than a fixed partial order. In this language, the basic constraints are equality, and order $\prec_{mc}$. $X \prec_{mc} Y$ if and only if X is less than Y according to some partial order on D. Compound operations can be formed by taking conjunctions or disjunctions of other constraints.

The notions of satisfiability and implication are defined as follows. If C is a monotonicity constraint, and $<$ is a fixed partial order on D, we let $C_<$ denote the partial order constraint obtained by replacing $\prec_{mc}$ with $<$ in C. We say C is *satisfiable* if for some partial order $<$, the partial order constraint $C_<$ is satisfiable. A set of monotonicity constraints is satisfiable if the conjunction of constraints in the set is satisfiable.[1] We say $C1 \models C2$ if for every partial order $<$, $C1_< \models C2_<$. □

Monotonicity constraints are more general than partial order constraints because they allow one to specify that a relation is acyclic without specifying the underlying order in advance.

Example 3.4: (Arithmetic Constraints) Let D be the integers, with terms expressing addition and multiplication of numbers. Then, with equality and inequality constraints we can express arithmetic constraints such as $X < Y * Y + 3$ and so on. With an existential quantifier we can express the evenness of X by using the formula $\exists Y(X = Y + Y)$. Arithmetic constraints can also be applied to the rationals or the reals. □

For further examples of constraint domains, see [JM94].

3.1 Projection

Definition 3.3: Let C be a constraint and let S be the set of variables appearing in C. Let $S' \subseteq S$. The *projection* C' of C onto S' is a constraint with variables in S' having the following properties:

- $C \models C'$; and if E is another constraint on the variables in S' such that $C \models E$, then $C' \models E$.

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. The definition of projection implies that when the projection exists it is unique up to equivalence.

The projection C' of C onto S' is said to be *exact* if for every substitution σ for the variables in S', $C'\sigma$ is true precisely when there exists a substitution τ for the variables in $S - S'$ such that $C\sigma\tau$ is true. □

The definition of an exact projection could intuitively be written as $C' \models \exists \vec{s} C$ where $\vec{s}$ represents the variables in $S - S'$. If the constraint language allows existential quantifiers, then such a formula makes sense, and projections will always be exact. However we cannot assume in general that the constraint language allows existential quantifiers.

For some constraint languages without existential quantifiers, such as monotonicity constraints [BS89], one can show that the projection of a constraint always exists and is exact. However, this property is not true in general.

[1] Note that this is not the same as saying that every element of the set is satisfiable, because elements of the set could be satisfiable with respect to different partial orders.

Example 3.5: Consider the constraint

$$(X < Y) \wedge (Y < Z) \wedge (Y = W + W)$$

in a language over the nonnegative integers with no quantifiers. The projection onto $\{X, Z\}$ "should" be

$$\exists Y, W : (X < Y) \wedge (Y < Z) \wedge (Y = W + W)$$

except that our language doesn't allow quantifiers. The constraint $X+1 < Z$ is certainly implied by our original constraint, but it does not contain all the information present in the initial constraint. $X + 1 < Z$ fails to qualify as an exact projection since it does not require that there is an even number between X and Z. For this example there is no exact projection. □

Since we would like to get *some* information when we restrict a constraint to a subset of its variables, even when projections do not exist, we will sometimes be happy with a weaker notion, that of a pseudo-projection.

Definition 3.4: Let C be a constraint and let S be the set of variables appearing in C. Let $S' \subseteq S$. A *pseudo-projection* C' of C onto S' is a constraint on the variables in S' such that $C \models C'$. □

Any tautological constraint is a pseudo-projection, so it is clear that a pseudo-projection is weak. However, in some cases we can find pseudo-projections that contain some, if not all of the information embodied in the original constraint. In Example 3.5 above, $X + 1 < Z$ and $X < Z$ are both pseudo-projections of the original constraint. Even when projections are guaranteed to exist, they may take too long to compute. At the price of obtaining a weaker formula, one may settle for a pseudo-projection if it can be obtained more quickly. This is an example of a more general kind of approach of doing approximate reasoning to avoid potential problems of intractability [DE92].

Since there may be many pseudo-projections, we shall assume that an algorithm is given that, given a constraint C and a set of variables S', will choose a pseudo-projection of C onto S'.

3.2 Inferring Constraints on IDB Predicates

Given constraints on the EDB predicates, it may be possible to infer constraints on the IDB predicates. For example, if the edge relation e of a graph satisfies the monotonicity constraint $e(X,Y) : X \prec_{mc} Y$, then one can infer using techniques from [BS89] that the transitive closure t of e satisfies $t(X,Y) : X \prec_{mc} Y$.

For monotonicity constraints on datalog programs, one can show that this inference problem is decidable [BS89]. However, for a different constraint language, the inference problem may not be decidable. Even for monotonicity constraints on programs with stratified negation, the inference problem is undecidable [Ros94b]. Hence we must be satisfied in general with a *partial* inference, i.e., an inference algorithm that is sound but not necessarily complete. We assume that, for any constraint language, a partial inference algorithm is given. We shall call a set S of constraints *sound* if the constraints in S on the IDB predicates can be inferred from the constraints in S on the EDB predicates using the given sound inference algorithm.

3.3 Implication and Unsatisfiability

For some constraint domains it is possible to algorithmically determine whether one constraint formula implies another. However, there are some constraint domains for which determining whether one constraint formula implies another is an undecidable problem. For such domains, we require that an algorithm be given that is sound, i.e., it determines whether F_1 implies F_2 some but not all of the time. We also require that any such algorithm be able to correctly identify that F_1 implies F_2 when F_2 belongs to a specified list of tautological constraints.[2]

We shall need an algorithm to determine whether a constraint formula is unsatisfiable. In general, determining unsatisfiability in an arbitrary constraint domain might not be decidable. Hence we shall settle for an algorithm that detects unsatisfiability some, but not all of the time. Such an algorithm would try to determine that a formula is inconsistent.

3.4 Constraints on Rules

We shall be interested in allowing constraints on rules in addition to constraints on predicates. We can infer a constraint on rule r from a constraint set E on the predicates appearing positively in the body of r. For example, given constraints

$$\begin{array}{l} p(X,Y) : Y \prec_{mc} X \\ q(X,Y) : Y \prec_{mc} b \\ t(X) : X = a \\ s(X,Y,Z) : Y = Z \end{array}$$

and rule r with body $p(W,X), q(X,Y), t(Y), s(Y,a,V)$ we can infer the constraint

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

There is a simple way to perform this inference: simply conjoin the constraints on the individual atoms. (Recall that a constraint language must be closed under conjunction.) We shall assume that we have performed this inference for every rule r. We also want to "go the other way," projecting a constraint on a rule onto the arguments of some of its atoms.

[2] This requirement ensures that when we eventually widen a formula to a tautology (as discussed later), we ensure that the widened formula is seen to be implied by other formulas.

Definition 3.5: Let r be a rule, and let E 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 $p_h{:}1, \ldots, p_h{:}j$.

Suppose that $r : C$ is the constraint on rule r. Append (i.e., conjoin) to 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.

We (pseudo-)project the above constraint onto the variables $\{q_b{:}1, \ldots, q_b{:}m, p_h{:}1, \ldots, p_h{:}j\}$ to get a new constraint C'. We call C' the *(pseudo-)projection* of C onto (p_h, q_b), denoted $\pi_E(r, p_h, q_b)$. □

Example 3.6: Consider a constraint domain over the integers, without any quantifiers. Let r be the rule

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

and suppose we have the constraint C on rule r given by $r : (X < W) \wedge (Y < 100) \wedge (Y = Z + Z)$. We project r onto (t_h, s_b). After appending the conjunction to C we have

$$(X < W) \wedge (Y < 100) \wedge (Y = Z + Z) \wedge \\ (t_h{:}1 = W) \wedge (t_h{:}2 = Y) \wedge (s_b{:}1 = Y).$$

After the pseudo-projection we get

$$(t_h{:}2 < 100) \wedge (t_h{:}2 = s_b{:}1).$$

Since we don't have existential quantifiers, we lose the evenness constraint ($t_h{:}2$ must be even) in the pseudo-projection. □

4 Universal Constraint Stratification

In this section we define a stratification condition that is general enough to include all of the examples from Section 2.1 (and also those in [Ros94a]), while also including some programs that are not modularly stratified. In a later section, we shall demonstrate that the condition is syntactic. Our condition can be phrased for any constraint domain, unlike [Ros94b] which considers only monotonicity constraints.

The intuition behind our stratification condition is as follows. We shall start with a set of constraint formulas describing the EDB of a program. In general, we may also infer some predicate constraints for the IDB relations, as discussed in Section 3.2. 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.* For finite databases, this will ensure a two-valued well-founded model. We must consider both direct recursion through negation, and recursion through negation via other mutually recursive predicates.

Definition 4.1: Let E 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 E*, whose nodes are the recursively-defined predicates from P. Arcs in the predicate constraint graph have two properties: a polarity, either positive or negative, and a constraint label. There is an arc from q to p with positive polarity if

$$p(\cdots) \leftarrow \cdots, q(\cdots), \cdots$$

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

$$p(\cdots) \leftarrow \cdots, \neg q(\cdots), \cdots$$

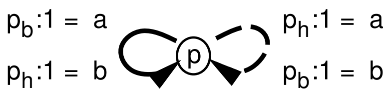
is a rule in P and p and q are mutually recursive. An arc for rule r is labeled with $\pi_E(r, p_h, q_b)$. □

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 when the meaning is clear. Our notion of predicate constraint graph is slightly different from that of [Ros94b] in that we omit nonrecursive predicates from the graph. This difference is immaterial for the definition of universal constraint stratification, but it will have some technical advantages when talking about structural totality in Section 5.

Example 4.1: The program P_1 given by

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

with no additional constraints, has the predicate constraint graph



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 $e(X, Y) : (X \prec_{mc} Y) \vee (X = Y)$ being the single monotonicity constraint. The predicate constraint graph is given in Figure 1. □

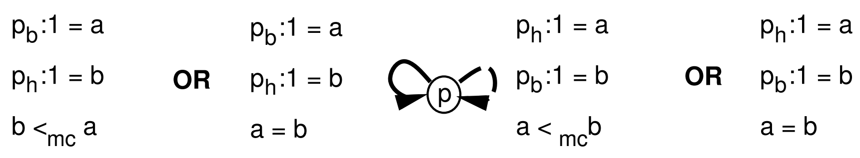


Figure 1: Constraint graph from Example 4.1.

4.1 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 4.2: (Sequential Composition) Let E be a set of constraints on predicates of a program P. Let $C_1 = \pi_E(r_1, p_h, q_b)$ and $C_2 = \pi_E(r_2, q_h, t_b)$ be constraints. Let j, m, k be the respective arities of p, q, t. The *sequential composition* of C_1 and C_2, denoted $\pi_E(r_1 \circ r_2, p_h, t_b)$, 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 C_1 and C_2. The sequential composition is the (pseudo-)projection onto $\{p_h{:}1, \cdots, p_h{:}j, t_b{:}1, \cdots, t_b{:}k\}$ of

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

The sequential composition of a sequence of constraints is defined as the successive pairwise sequential composition of its elements. If we always obtain the exact projection, then sequential composition is associative (up to equivalence); otherwise, the result of the sequential composition depends on the order chosen. In that case, we assume that a fixed algorithm is given for choosing the order in which to apply a sequence of sequential compositions. □

Definition 4.3: (Cyclic Composition) Let E be a set of constraints. Let $\pi_E(r_1, p_h^1, p_b^2)$, $\pi_E(r_2, p_h^2, p_b^3)$, $\ldots$, $\pi_E(r_n, p_h^n, p_b^1)$ 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 $\theta_E(r_1, \cdots, r_n)$ and is defined as the (pseudo-)projection onto the empty set of variables of the formula

$$\pi_E(r_1 \circ \cdots \circ r_n, p_h^1, p_b^1) \wedge$$
$$(p_h^1{:}1 = p_b^1{:}1) \wedge \cdots \wedge (p_h^1{:}k = p_b^1{:}k)$$

where the arity of p^1 is k. □

[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 4.2: Consider the program P_1 from Example 4.1. The sequential composition $\pi_\emptyset(r_1 \circ r_2, p_h, p_b)$ is the projection of $(p_h{:}1 = b) \wedge (p_b'{:}1 = a) \wedge (p_h'{:}1 = a) \wedge (p_b{:}1 = b) \wedge (p_b'{:}1 = p_h'{:}1)$ onto (p_h, p_b). Note the renaming of the intermediate p to p'. The result is $(p_h{:}1 = b) \wedge (p_b{:}1 = b)$. The cyclic composition is then the projection of $(p_h{:}1 = b) \wedge (p_b{:}1 = b) \wedge (p_h{:}1 = p_b{:}1)$ onto the empty set of variables, which can be represented as the trivially satisfiable constraint $b = b$.

Consider the program P_2 from Example 4.1, and let C be the monotonicity constraint given there for e. The sequential composition $\pi_C(r_1 \circ r_2, p_h, p_b)$ is equivalent to

$$(p_h{:}1 = a) \wedge (p_b{:}1 = a) \wedge [(a = b) \vee (a = b \wedge a \prec_{mc} b) \vee$$
$$(a = b \wedge b \prec_{mc} a) \vee (a \prec_{mc} b \wedge b \prec_{mc} a)].$$

Observe that the disjunction above is unsatisfiable, since $a \not\equiv b$, and since we cannot simultaneously have $a \prec_{mc} b$ and $b \prec_{mc} a$. The cyclic composition is thus also unsatisfiable, and can be represented as the trivially unsatisfiable constraint $a \prec_{mc} a$. □

4.2 Checking Cyclic Compositions

In the next section we shall define the notion of universal constraint stratification in terms of the predicate constraint graph having the following property: all cycles with a negative edge have an unsatisfiable cyclic composition. In this section we address the issue of how to determine this property given that there may be infinitely many such cycles, even for simple programs.

The cyclic composition θ is formed by taking a sequential composition π and adding some constraints. π may be replaced by any equivalent (or weaker) constraint and yield an equivalent (or weaker) 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. In order to avoid inferring an infinite sequence of constraints, none of which is implied by previous constraints, we employ *widening* [CC77]. Widening progressively weakens newly generated constraints in the hope that eventually the system will detect that the weakened constraint is implied by previous constraints.

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 that contain a negative edge, and calculate the sequential composition on those cycles.

2. For each node in the predicate constraint graph, construct all cycles with a negative edge having exactly one sub-cycle, and calculate the sequential composition on those cycles. If at least one of these cycles yields sequential composition not implied by previous sequential compositions 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. If the iteration does not converge after a fixed number of steps, apply widening to weaken the newly derived sequential compositions. By applying widening in a way that is known to guarantee termination (eg., by weakening to a tautology within a finite number of steps) one can guarantee that the process above will terminate.

Suppose the process above terminates, and finds that all cycles do have unsatisfiable cyclic compositions. In this case, since the widened compositions are weaker than the "real" compositions, one can argue that all cycles are in fact unsatisfiable.

On the other hand, widening may cause us to miss some constraint graphs that do satisfy the property of unsatisfiable cycles. We may widen to the point that an unsatisfiable constraint becomes satisfiable. For many constraint domains, it is impossible to determine the property of unsatisfiable cycles exactly, and so we must let sufficient conditions suffice. In the next section we shall assume that an algorithm for applying the steps outlined above is given.

In some cases (such as for monotonicity constraints) one can check all cycles for unsatisfiable cycles *exactly*. The important observations are that (a) For a given program P and a given constraint domain D, there are finitely many nonequivalent monotonicity constraints on D given a fixed set of variables, and (b) One can algorithmically determine whether two monotonicity constraints are equivalent [BS89].

4.3 Exactness

So far there have been several sources of approximate reasoning: (a) Pseudo-projections rather than projections, or non-exact projections, (b) Incomplete implication tests, (c) Incomplete unsatisfiability checking, and (d) Widening of generated constraints. For some constraint domains, it is possible to avoid all of these approximations, and algorithmically (a) generate exact projections, (b) test implication exactly, (c) test unsatisfiability exactly, and (d) avoid widening altogether because there are only finitely many inequivalent constraints for a given finite set of constants and variables. We shall call such a constraint domain *exact*.

Lemma 4.1: Let D be an arbitrary finite nonempty set of constants. The following constraint languages, together with D, form exact constraint domains: (a) Equality constraints, (b) Equality constraints with disjunction, (c) Monotonicity constraints, and (d) Monotonicity constraints with disjunction. ∎

One example of a constraint domain that is not exact is the domain of partial order constraints over a finite nonempty domain. Projections on this domain are not exact. For example, there is no way to exactly project $(X < Y) \wedge (Y < Z)$ onto X and Z, since without existential quantifiers there is no way to capture the constraint that there must be an intervening element between X and Z.

While many constraint domains admit exact projections, exact implication tests and exact unsatisfiability testing, there are fewer that satisfy the condition of having only finitely many inequivalent constraints. Almost any constraint domain with arithmetic fails this last condition because there will be infinitely many inequivalent constraints, such as $Y = X$, $Y = X + X$, $Y = X + X + X$, etc.

4.4 Universal Constraint Stratification

Definition 4.4: Let F be a constraint domain, let E be a sound set of constraints from F on the predicates in P, and let G be the predicate constraint graph of P with respect to E. We say that P is *universally constraint stratified with respect to* E and F if for every cycle in G that contains a negative arc, the cyclic composition of the constraints on the arcs in the cycle is unsatisfiable. □

(Note that in the case that pseudo-projections are taken, widening is applied, or if we have an incomplete algorithm for determining unsatisfiability, then there is an implicit dependence on the algorithms used for the pseudo-projection, the sequential composition, widening, and the unsatisfiability test.)

Example 4.3: Consider once more the program P_1 from Example 4.1. As we saw in Example 4.2, P_1 has a cycle with a negative arc whose cyclic composition yields a satisfiable constraint. Thus P_1 is not universally constraint stratified with respect to the empty set of constraints and the domain of equality constraints. □

Example 4.4: Given $C = \{part(X,Y) : (Y \prec_{mc} X)\}$ for the constraint domain M of monotonicity constraints, the program of Example 2.1 has predicate constraint graph given in Figure 2, and is universally constraint stratified with respect to C and M since (as

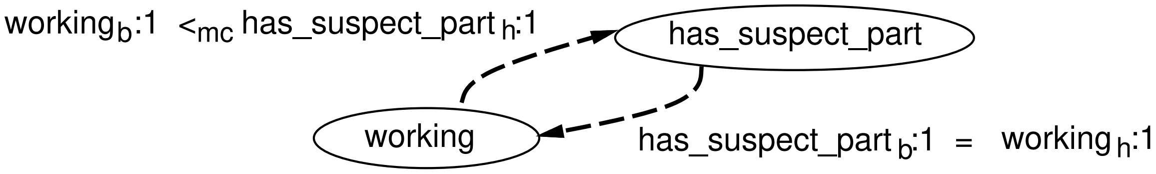
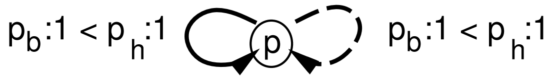


Figure 2: Constraint graph for the *working* program in Example 4.4.

the reader may verify) the cyclic composition of the constraints on the cycles is unsatisfiable. □

Example 4.5: Suppose that we have the constraint set $C = \{e(X,Y,Z) : (X = YZ) \wedge (X > 1) \wedge (Y > 1)\}$ from the constraint domain I^* of integer constraints with multiplication. The program of Example 2.2 has predicate constraint graph



with respect to C. We pseudo-projected the constraint

$$(X = YZ) \wedge (X > 1) \wedge (Y > 1) \wedge (p_h{:}1 = X) \wedge (p_b{:}1 = Y)$$

onto $\{p_h{:}1, p_b{:}1\}$ to get $p_b{:}1 < p_h{:}1$ and similarly for the other occurrence of p in the body of the third rule. Let us examine in detail how we might determine that all cyclic compositions are unsatisfiable. We first calculate the sequential compositions on all simple cycles containing a negative edge. There is only one such cycle, with the constraint $p_b{:}1 < p_h{:}1$. The cyclic composition for this cycle is

$$(p_b{:}1 < p_h{:}1) \wedge (p_b{:}1 = p_h{:}1)$$

which is unsatisfiable. We then look for cycles with one sub-cycle and at least one negative edge. There are two such cycles. In both cases the sequential composition of the constraints is $p_b{:}1 < p_h{:}1$. This is exactly the same sequential composition we found earlier; let us assume that the equivalence with the previous constraint is determined by our (possibly partial) equivalence-testing algorithm. Since we found no new sequential compositions at this point we can stop and conclude that the cyclic composition of the constraints on each cycle is unsatisfiable. Thus, this program is universally constraint stratified with respect to C and I^*. □

Example 4.6: Given the constraints $C = \{s(X,Y,Z) : (X = Y + Z) \wedge (1 \leq X \leq n) \wedge (1 \leq Y \leq n) \wedge (1 \leq Z \leq n), b(X,Y,Z) : X < Y < Z\}$ from the constraint domain I^+ of integer constraints with addition, the program of Example 2.3 has predicate constraint graph given in Figure 3. We pseudo-projected the constraints of *fib* and *between* to obtain the given constraints. This program is universally constraint stratified with respect to C and I^+ assuming that our algorithm for testing unsatisfiability notices that the cyclic composition of the constraints on the cycles is unsatisfiable. □

So far, every universally constraint stratified program we have seen has been modularly stratified. It is shown in [Ros94b] that there exist universally constraint stratified programs that are not modularly stratified. Conversely, there are modularly stratified programs that are not constraint stratified. Due to lack of space we omit these examples here.

Theorem 4.2: Let D be a constraint domain, and let C be a sound set of constraints from D on a program P. Let E be the constraints from C on EDB predicates. If P is universally constraint stratified with respect to C and D, and if the EDB satisfies E, then P has a two valued well-founded model. ∎

The converse of Theorem 4.2 is not true in general.

The following result shows that local stratification (for a function-free program) is exactly the same concept as a version of universal constraint stratification. Let H denote the Herbrand universe of a program, i.e., the set of constants appearing in the program, and let F be a constraint language involving equalities of terms in H (and conjunctions of such equalities). Lemma 4.1 shows that this constraint domain is exact. (A formula is unsatisfiable if and only if it equates two constants, either directly or transitively through some intermediate variables.) Assuming that we use these exact tests, we obtain the following result.

Theorem 4.3: Let F and H be as defined above. A program is universally constraint stratified with respect to the empty set of constraints and (F, H) if and only if it is locally stratified. ∎

As a consequence of Theorem 4.3, local stratification can be determined for function-free programs without actually instantiating the whole program.

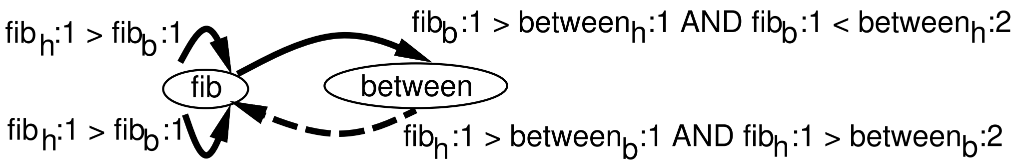


Figure 3: Constraint graph for the *Fibonacci* program in Example 4.6.

4.5 Universal Constraint Stratification is Syntactic

We say that a condition on deductive databases 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 is updated. 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 decidable.

We need to determine that all of the steps used to define universal constraint stratification can be derived algorithmically. These transformations include inferring constraints, performing projections, performing compositions, and testing all cycles for unsatisfiability. That each of these steps is algorithmic follows from the fact that in each case we have opted (if necessary) to settle for an algorithm that is incomplete but effective. Thus, universal constraint stratification is syntactic.

5 Structural Totality

Papadimitriou and Yannakakis have proposed a notion of structural totality [PY92]. Several variants of structural totality were presented in [PY92]; we focus particularly on one of these variants.

Definition 5.1: A program P is *structurally well-founded total in the non-uniform sense* if for every program Q with the same skeleton as P, and for every possible set Δ of EDB relations for Q, $Q \cup \Delta$ has a two-valued well-founded model. For brevity, we shall abbreviate "structurally well-founded total in the non-uniform sense" to "structurally total." □

The motivation for defining structural totality was to identify programs that had two-valued well-founded models for *structural* reasons, and not because of some "intricate pattern in which variables and constants repeat in the rules." Thus, all possible programs with the same skeleton are considered, rather than just the given program. As an additional motivation, it is proved in [PY92] that determining whether a given program has a two-valued well-founded model for all EDBs is an undecidable problem.

The following theorem is presented in [PY92].

Theorem 5.1: A useless-free program P is structurally total if and only if it is stratified. ▮

One way to interpret this theorem is that the only programs that have two-valued models *for structural reasons* are the stratified programs. All other programs with two-valued well-founded models are two-valued because of some pattern of variable and constant bindings in the rules.

In this paper we argue that one does not have to restrict one's attention to stratified programs in order to get a syntactic, structural condition that guarantees a two-valued well-founded model. In the general case, there may be an "intricate pattern in which variables and constants repeat in the rules." However, we identify a significant class of programs for which one can, in essence, syntactically identify a "good" passing of bindings. We can thus handle many natural nonstratified programs.

5.1 Extending Structural Totality

In this section we extend the notion of structural totality and prove a result analogous to Theorem 5.1 in the more general case. Rather than comparing the *skeletons* of programs, we compare their *constraint graphs.*

Definition 5.2: Two constraint graphs are said to be *equivalent* if they have the same set of nodes, the same set of edges, and if the constraints on corresponding edges are equivalent. □

Definition 5.3: Let P be a program, let C be a sound constraint set for P, and let G be the predicate constraint graph of P with respect to C. We shall say that P is structurally total with respect to C if for every program Q and sound constraint set C_Q such that Q has a predicate constraint graph with respect to C_Q that is equivalent to G, the following holds: For every possible set Δ of EDB relations (for Q) satisfying C_Q, $Q \cup \Delta$ has a two-valued well-founded model. □

Theorem 5.2: Let P be a program and C a sound set of constraints for P from an exact constraint domain F. Then P is structurally total with respect to C if and only if P is constraint stratified with respect to C and F. ▮

This result is important because it demonstrates that, for exact constraint domains, it is possible to get beyond Theorem 5.1 to identify nonstratified programs that are still well-behaved "for structural reasons." We have extended the concept of structure to include not just predicate calling patterns, but also the constraints between arguments that hold because of schema-level integrity constraints. Papadimitriou and Yannakakis exclude reasoning about the arguments of predicates because they wanted to ignore the "intricate pattern in which variables and constants repeat in the rules." While it is true that in the general case the pattern of constants and variables in the rules can be very intricate, we believe that at least some of the time this pattern is highly ordered, and is identifiable on the basis of some schema-level integrity constraints.

6 Conclusions and Further Work

In [Ros94b] the present author defined the notion of "universal constraint stratification." That notion is more restricted than this paper's version of universal constraint stratification because the former considers only monotonicity constraints. In this paper we consider an arbitrary constraint domain, and are not tied to one specific formalism. One can "plug in" an appropriate constraint domain and obtain the corresponding notion of universal constraint stratification. Further, if one can prove that one's constraint domain is exact, then one can use Theorem 5.2 to demonstrate that this condition is a natural condition identifying the "structurally good" programs.

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, (d) Admits the examples from [Ros94a] and Section 2.1 of useful programs that are not locally stratified, (e) Can be parameterized by an arbitrary constraint domain, and (f) Has a clear structural interpretation for exact constraint domains.

We can slightly modify our condition of universal constraint stratification to require that all cycles, not just those having at least one negative edge, have unsatisfiable constraints. With that extension we get a condition analogous to "acyclicity" [AB90] or "modular acyclicity" [Ros91]. Programs satisfying these conditions have good termination properties: they can be evaluated either top-down *without memoing*, or bottom-up *without duplicate elimination*. In general, both top-down methods without memoing and bottom-up methods without duplicate elimination can get trapped in infinite loops even for programs with a finite semantics.

Universal constraint stratification applies also to programs with aggregation. In [Ros94b], the parts-explosion problem is formalized using recursion through aggregation, and a monotonicity constraint on the part hierarchy implies the constraint stratification condition. The work of this paper also applies to programs with aggregation. While [Ros94b] allowed only monotonicity constraints, we allow arbitrary constraint domains and could thus handle additional aggregate programs and constraints.

Programs like those of Example 2.3 could be expressed using constraints explicitly in the rules using a constraint logic programming language [JL87]. In general, one obtains higher expressive power with a constraint logic programming language, and so constraint logic programs form a natural class to study in order to find more general syntactic conditions for a two-valued well-founded model. In this paper we considered constraints available only as implicit restrictions on EDB atoms. One could extend our formalism to also incorporate explicit constraints into our constraints on rules. Assuming that we could demonstrate that the constraint logic program had a finite set of nonfalse (i.e., true or undefined) answers, the condition obtained would ensure a two-valued well-founded model.

Another important area for further work is developing algorithms for incrementally checking integrity constraints on the database for various constraint domains. We do not want to have to check the whole of an EDB relation each time a tuple is added or deleted. If possible we would like to only look at the newly added or deleted tuple together with a limited amount of extra information.

Acknowledgements

The author thanks Divesh Srivastava and Peter Stuckey, who introduced me to the concepts of widening and pseudo-projections, and suggested considering arbitrary constraint domains.

References

[AB90] K. R. Apt and M. Bezem. Acyclic programs.

In *Proceedings of the Seventh International Conference on Logic Programming*, 1990.

[BF91] N. Bidoit and C. Froidevaux. Negation by default and unstratifiable logic programs. *Theoretical Computer Science B*, 78(1):85–112, 1991.

[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.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 238–252, 1977.

[DE92] M. Dalal and D. W. Etherington. Tractable approximate deduction using limited vocabularies. In *Proceedings Ninth Canadian Conference on Artificial Intelligence (AI '92)*, pages 206–212, Vancouver, Canada, 1992.

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

[JL87] J. Jaffar and J. L. Lassez. Constraint logic programming. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 111–119, 1987.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19,20:503–581, 1994.

[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.

[Prz89] T. C. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *ACM Symposium on Principles of Database Systems*, 1989.

[PY92] C. Papadimitriou and M. Yannakakis. Tie-breaking semantics and structural totality. In *Proceedings of the Eleventh ACM Symposium on Principles of Database Systems*, pages 16–22, June 1992.

[Ros91] K. A. Ross. *The Semantics of Deductive Databases.* PhD thesis, Stanford University, August 1991. Stanford University Technical Report STAN-CS-91-1386.

[Ros94a] K. A. Ross. Modular stratification and magic sets for Datalog programs with negation. *Journal of the ACM*, 41(6):1216–1266, 1994.

[Ros94b] K. A. Ross. A syntactic stratification condition using constraints. In *Proceedings of the International Symposium on Logic Programming*, November 1994.

[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. The well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.