1. INTRODUCTION

Datalog with negation is a very natural and well-studied database language. The question of assigning appropriate semantics to Datalog programs with negation, which we call the tie-breaking semantics. The tie-breaking semantics can be computed in polynomial time and results in a fixpoint whenever the rule-goal graph of the program has no cycle with an odd number of negative edges. We show that, in some well-defined sense, this is the most general fixpoint semantics of negation possible; in particular, we show that if a cycle with an odd number of negative edges is present, then the logic program is not structurally total, that is, it has an alphabetic variant which has no fixpoint semantics whatsoever. Determining whether a program is total is undecidable.

We address the question of when the structure of a Datalog program with negation guarantees the existence of a fixpoint. We propose a semantics of Datalog programs with negation, which we call the tie-breaking semantics. The tie-breaking semantics can be computed in polynomial time and results in a fixpoint whenever the rule-goal graph of the program has no cycle with an odd number of negative edges. We show that, in some well-defined sense, this is the most general fixpoint semantics of negation possible; in particular, we show that if a cycle with an odd number of negative edges is present, then the logic program is not structurally total, that is, it has an alphabetic variant which has no fixpoint semantics whatsoever. Determining whether a program is total is undecidable.

The well-founded semantics proposed a few years ago and described in [VRS] (see Section 2 for the formal definition) coincides with stratification on all stratified programs, but it also succeeds in finding a fixpoint of certain unstratified programs, such as

\[ P(a) \leftarrow \neg P(x), E(b). \] (1)

The well-founded semantics constructs a (possibly partial) model by considering the ground graph, whose vertices are ground atoms and whose directed edges reflect the rules. This semantics is defined for all programs with negation, although it does not always succeed in constructing a fixpoint (a total model). It was left open in [VRS] to characterize the programs for which the well-founded semantics succeed in constructing a fixpoint (that is, when all ground atoms end up with a truth value, for all databases); also, it was proposed as a “final frontier” challenge to define even more general semantics.

A related semantics proposed around the same time is that based on the notion of a default or stable model [BF1, GL]. A comprehensive survey of the different approaches to negation in logic programming can be found in [Bi]. Furthermore, several recent papers explore the relationships between these different semantics [BF2, Du, Gi, MS].

Fixpoint semantics of negation (otherwise known as “models of the Clark extension”) were studied from the computational point of view in [KP]. In that paper certain complexity-theoretic obstacles to this approach were pointed out; e.g., it is NP-complete to tell if a program has a fixpoint on a given database. However [KP] did not consider the same question for all databases, an important issue in the context of datalog. Let us call a logic program with negation total if it has at least one (and possibly more) fixpoint for all databases (that is, when it is indeed a total, perhaps multivalued, function from databases to

\[ \text{of the form } Q \leftarrow \ldots, P\ldots, \text{ and a negative edge if it has a rule } Q \leftarrow \ldots, \neg P\ldots. \]

\[ P(a) \leftarrow \neg P(x), E(b). \] (1)
TIE-BREAKING SEMANTICS

From the above discussion we can distinguish two aspects of the same important open question in the theory of Datalog: First, is there a polynomially computable fixpoint-based semantics that is more general than the well-founded one? And if so, is there one that is the most general? Put otherwise, what makes a Datalog program with negation total, and if it is total, how fast can we compute a fixpoint?

In this paper we address these questions, and largely answer them in a rather unexpected and satisfactory way. We extend the well-founded semantics to define what we call the tie-breaking semantics. In contrast to the well-founded semantics, the tie-breaking semantics is nondeterministic, in that the algorithm that defines and calculates it makes arbitrary choices that affect the final fixpoint. It is open to opinion and debate whether such a semantics can be natural and useful. It should be noted however that allowing arbitrary choices is not as strange as it may sound at first, and in fact other authors have argued in the past that nondeterminism has a useful role to play and have proposed the incorporation of such a feature in the language (see, e.g., [AV, KN]). Sacca and Zaniolo show in [SZ] how the stable model semantics can be used to express nondeterminism and study this interplay in more depth.

In this paper we argue that the tie-breaking semantics is in a certain well-defined “structural” sense the most general possible fixpoint-based semantics of Datalog with negation. To understand why, let us first point out that a sufficient condition for the tie-breaking semantics to compute a fixpoint is that the graph of the program contains no cycle with an odd number of negative edges. This class of programs and the fact that they always have a fixpoint were first identified by Kunen in [Ku]; also, a version of the tie-breaking semantics was proposed in [PS] as an extension-finding mechanism in the context of default logic. Furthermore, and perhaps more interestingly, we can show that this is an essentially necessary condition for the program to be total.

Logic programs may have a terrible cycle structure and still be total due to the intricate pattern in which variables and constants repeat in the rules (recall program (1) above; more complex examples abound). We prove that, testing a program for totality is an undecidable problem in both the uniform and the nonuniform case (Theorem 6).

Define now two programs to be alphabetic variants of one another if they become the same if we only pay attention to the predicate symbols in the rules and ignore the arguments; i.e., if the two programs only differ in the arity of the predicates and the names of the variables and constants in each rule. For example,

\[ P(x, y) \leftarrow \lnot P(y, y), E(x). \]  

(2)
is an alphabetic variant of program (1) above. And define now a program to be structurally total if all of its alphabetic variants are total, i.e., have at least one fixpoint. For example program (1) is total but not structurally total, since its alphabetic variant (2) is not total (it has no fixpoint whenever \( E \) is nonempty). Structural totality is a natural way of formalizing the concept of programs that always have a fixpoint due to their structure, while disregarding pathological programs that are total due to the unpredictable intricacies of the way in which the detailed interaction of the rules impedes the transmission of information. Our evidence that tie-breaking semantics is the most general fixpoint semantics of Datalog programs with negation is our Theorem 2, stating that a program is structurally total if and only if its program graph has no cycle with an odd number of negative edges. An analogous, technically harder result holds in the non-uniform case (where the IDB’s are assumed to be initialized to empty, Theorem 3). It follows from our results that testing a Datalog program for structural totality can be done in linear time and in NC, whereas in the nonuniform case it is still linear time but \( P \)-complete (Theorem 4). In fact, our proofs do not need the full power of structural totality, in that we construct alphabetic variants consisting of only binary predicates. Finally, we show in Theorem 5 that stratified programs are precisely those for which the well-founded semantics finds a fixpoint, independently of input database and alphabetic variant.

The rest of the paper is organized as follows. In the next section we review briefly basic definitions and notation and describe fixpoints, the stable model, and the well-founded semantics. In Section 3 we define the tie-breaking semantics and state some of its properties. Section 4 contains the characterizations of programs that are structurally total as described above, and Section 5 proves the undecidability of determining totality of a program. Section 6 contains concluding remarks and open problems.

2. PRELIMINARIES

First we review briefly basic definitions and notation. The reader is referred to [Ul] for more information on Datalog and to [Bi] for negation. If \( P \) is an \( m \)-ary predicate symbol and \( x_1, \ldots, x_m \) are (not necessarily distinct) variables or constants, then \( P(x_1, \ldots, x_m) \) is called an atom; it is ground if all the arguments are constants. A literal is an atom \( P(x_1, \ldots, x_m) \) or the negation of an atom \( \lnot P(x_1, \ldots, x_m) \).
A Datalog program with negation $\Pi$ is a finite set of rules of the form

$$A \leftarrow L_1, \ldots, L_r,$$

where $A$ is an atom (positive literal) called the head of the rule, and $L_1, \ldots, L_r$ are (positive or negative) literals forming the body of the rule.

We are given a Datalog program with negation $\Pi$. We distinguish between the extensional database (EDB) predicates of $\Pi$ that do not appear at the head of any rule, and the intentional database (IDB) predicates that do. Let $A$ be an initial (finite) database, i.e., a set of initial values for all predicates (relations) of $\Pi$. Let $G$ be the universe including all constant symbols appearing in $\Pi$ and $A$, and let $V_P$ be the set of all ground atoms over $G$. That is, for each $m$-ary predicate $Q$ of $\Pi$ and for each $m$-tuple $(a_1, \ldots, a_m)$ of constant symbols from $U$, we have $Q(a_1, \ldots, a_m) \in V_P$.

We define the ground graph of $\Pi$ and $A$, a bipartite directed graph denoted $G(\Pi, A) = (V_P, V_R, E_+, E_-)$. The graph has two disjoint sets of edges, $E_+$ and $E_-$ (the positive and negative edges), and two kinds of nodes, predicate nodes $(V_P)$ and rule nodes $(V_R)$. The set of predicate nodes is the set $V_P$ of ground atoms over the universe $U$. For each rule $r$ of $\Pi$ involving $k$ variables and for each $k$-tuple $(a_1, \ldots, a_k)$ of constant symbols in $U$, we add a node $r(a_1, \ldots, a_k) \in V_R$, and the following edges: First, a positive edge from $r(a_1, \ldots, a_k)$ to $P(b_1, \ldots, b_m)$, where $P$ is the head (left-hand) predicate of $r$, and the $b_i$'s are the arguments of $P$ when the $a_i$'s are substituted for the variables of $r$. For each positive occurrence of a predicate symbol $Q$ in the body (the right-hand side) of $r$, we add to $E_+$ the edge from $Q(b_1, \ldots, b_m)$ to $r(a_1, \ldots, a_k)$, where the $b_i$'s are the arguments of $Q$ in the particular positive occurrence when the $a_i$'s are substituted for the variables. Finally, for each negative occurrence of a predicate symbol $Q$ in the body of $r$, we add the corresponding edge to $E_-$. The construction applies even if $k = 0$ (the rule has no variables), in which case there is exactly one rule node corresponding to the rule, and incident edges as specified above. This completes the construction of the ground graph $G(\Pi, A)$.

A (partial) model $M$ of $\Pi, A$ is a (partial) function from $V_P$ to the values true and false; the model is total if every atom of $V_P$ receives a truth value. We say that a ground literal $L$ is true in a partial model $M$ if it is a true atom or it is the negation of a false atom. The initial database $A$ corresponds to the partial model $M_0(A)$, where the value true is given to all EDB and IDB atoms explicitly concluded in $A$, and false to all EDB atoms not in $A$; the IDB atoms that are not in $A$ do not receive a truth value. A partial model $M_1$ extends another partial model $M_2$ if and only if every atom that has truth value in $M_2$ has also the same truth value in $M_1$. A partial model $M$ of $\Pi, A$ is consistent if it extends the initial model $M_0(A)$ and for every instantiated rule $r$, if all the literals in the body of $r$ are true under $M$, then also the head of $r$ is true.

A fixpoint of $\Pi$ for the input database $A$ is a total model $M$ in which an atom is true if and only if it belongs to the initial database $A$ or it is the head of an instantiated rule of $\Pi$ all of whose literals in the body are true under $M$. Every fixpoint is consistent, but not necessarily vice versa. Some authors use the term “supported model” for a fixpoint [ABW]. For a program $\Pi$ and input database $A$ there may be zero, one, or more fixpoints. Given $\Pi$ and $A$, it is NP-hard to tell if there is a fixpoint even in the propositional case, i.e., when all predicates have arity zero [KP].

The stable model semantics considers only some of the fixpoints as being “natural.” We will use here the ground graph to define this and other semantics. If $M$ is a partial model and $G$ is a ground, let close$(M, G)$ be the procedure that modifies $M$ and $G$ by applying the following operations repeatedly until they are inapplicable: If an atom $a$ is true in $M$, then it is deleted from $G$, and so are all rule nodes $r$ such that there is a negative arc $(a, r) \in E_-$; if an atom $a$ is false in $M$, then likewise, it is deleted from $G$, and so are all rule nodes $r$ such that there is a positive arc $(a, r) \in E_+$. If a rule node $r$ has no incoming edges, then the atom $a$ with $(r, a) \in E_+$ gets the value $M(a) := \text{true}$ and $r$ is deleted. If an atom $a$ has no incoming edge, then it gets the value $M(a) := \text{false}$. The procedure close$(M, G)$ returns the final model $M$ and graph $G$; it is easy to see that these are uniquely determined, independent of the order in which the modification rules are applied.

Let $\Pi$ be a program, $A$ an initial database, and $G$ the corresponding ground graph. Let $M$ be a total model that extends the initial partial model $M_0(A)$, and let $M_-$ be the partial model obtained from $M$ by letting all true IDB ground atoms that are not in $A$ be undefined instead of true (and the rest of the atoms keep the same truth value). Then, $M$ is a stable or default model if close$(M_-, G)$ reconstructs $M$, i.e., all undefined IDB atoms of $M_-$ become true [BF1, GL]. Every stable model is a fixpoint, but not conversely; i.e., some fixpoints may not be stable. For a program $\Pi$ and input database $A$ there may be zero, one, or more stable models. Given $\Pi$ and $A$, it is NP-hard to tell if there is a stable model even in the propositional case [Bi].

Fixpoint and stable model semantics are specified non-constructively, that is, they state desired properties of the intended model without saying how to actually compute it. In contrast, the well-founded semantics is constructive. An interpreter is an algorithm, which when given a program $\Pi$ and a database $A$, produces a partial model that extends $M_0(A)$. The well-founded semantics of $\Pi$ and $A$ can be computed by the following interpreter.

Let $G_+$ denote below the subgraph of the (current) ground graph $G$ consisting of the positive edges. If $M$ is a partial model, we use $\text{Atoms}[\text{close}(M, G_+)]$ to denote the
set of atoms that would be left in the graph if we applied the procedure \( \text{close}(M, G, 6) \). This set is the largest so-called unfounded set \([\text{VRS}]\). Graph-theoretically, a set of atoms \( D \) is unfounded, if the subgraph of \( G \) induced by the nodes of \( D \) and the rule nodes that precede them has no source (node with indegree 0).

**Algorithm Well-Founded.**

\[
M := M(\Pi, A); 
G := G(\Pi, A); 
(M, G) := \text{close}(M, G); 
\]
while \( C = \text{Atoms}[\text{close}(M, G)] \) is nonempty do:

- for each atom \( a \) in \( C \) define \( M(a) := \text{false} \);
- \( (M, G) := \text{close}(M, G) \)

It is easy to see that this algorithm runs in polynomial time. Van Gelder et al. prove that, when the computed model \( M \) is total (that is, the graph \( G \) has no atoms in the end), then \( M \) is a fixpoint of \( \Pi \) with \( A \); in fact, in this case there is a unique stable model, which is equal to \( M \) \([\text{VRS}]\).

### 3. Tie-Breaking Semantics

We define an interpreter, the *tie-breaking interpreter*, which computes a total model even in certain cases in which the well-founded one would not. We need a graph-theoretic interlude. Suppose that \( T = (V, E_+, E_-) \) is a strongly connected directed graph with positive and negative edges. We say that \( T \) is a *tie* if there is no cycle in it containing an odd number of negative edges; we call such a cycle *odd*. It does not matter in the definition whether we consider only simple cycles or arbitrary cycles, i.e., cycles that can go through the same node or edge more than once. The reason is that a general, nonsimple cycle \( C \) can be decomposed into a set of simple cycles; if \( C \) has an odd number of negative edges then this must be also the case for at least one of the simple cycles in the decomposition. These types of graphs were originally studied in graph theory by Harary in the 1950s who use the term *cycle-balanced*. Chapter 13 of \([\text{HNC}]\) contains a structural analysis including the following simple characterization.

**Lemma 1.** A strongly connected directed graph \( T = (V, E_+, E_-) \) is a *tie* if and only if its nodes can be partitioned into two sets \( K \) and \( L \), so that all positive edges stay within the partitions, and all negative edges cross partitions; i.e., \( E_+ \subseteq K \times K \cup L \times L \) and \( E_- \subseteq K \times L \cup L \times K \). We can test in linear time whether \( T \) is a *tie* and compute this partition if it is.

**Proof.** Starting from an arbitrary node \( u \), construct a directed spanning tree rooted at \( u \). Since \( T \) is strongly connected, \( u \) can reach every other node and hence, there is such a first search or depth-first search. Partition the nodes into two sets as follows. The root \( u \) is assigned arbitrarily to a set, say to \( K \). Every other node \( w \) is assigned arbitrarily to the same set as its parent if the arc from its parent is positive, and it is assigned to the opposite set if it is negative. It follows by a straightforward induction that a node \( w \) is assigned to set \( K \) (respectively \( L \)) if the unique path from \( u \) to \( w \) in the spanning tree has an even (resp. odd) number of negative edges.

Test now that all the arcs that are not in the tree are consistent with this partition, i.e., satisfy the conditions of the lemma. We claim that if some arc violates the conditions, then \( T \) is not a tie. Suppose that arc \((z, w)\) violates the condition. Consider the two paths from \( u \) to \( w \), one that goes from \( u \) to \( w \) within the spanning tree and the second that goes from \( u \) to \( z \) and then traverses the edge \((z, w)\). It is easy to see that the negative edges of the two paths have different parities. For example, if the arc \((z, w)\) is positive, node \( z \) is in \( K \) and \( w \) is in \( L \), then the first path has an odd number of negative edges and the second path has an even number. Since the graph is strongly connected, there is path from \( z \) to \( u \). Combining a \( w \rightarrow u \) path with one of the two \( u \rightarrow w \) paths will produce an odd cycle.

Thus, if the partition computed by the above algorithm does not satisfy the condition of the lemma, then \( T \) is not a tie. Conversely, if the partition satisfies the condition then \( T \) is obviously a tie.

We give first a “pure” version of the tie-breaking interpreter, and then we will describe a version that extends the well-founded semantics.

**Algorithm Pure Tie-Breaking.**

\[
M := M_0(\Pi, A); 
G := G(\Pi, A); 
(M, G) := \text{close}(M, G); 
\]
while there is a tie \( T \) in \( G \) with no incoming edges, do:

- let \((K, L)\) be the partition of \( T \) as in
  - Lemma 1 with \( L \) nonempty;
  - for each atom \( a \in K \) set \( M(a) := \text{true} \);
  - for each atom \( a \in L \) set \( M(a) := \text{false} \);
  - \( (M, G) := \text{close}(M, G) \).

It is easy to see that the tie-breaking interpreter is a polynomial-time algorithm. In an iteration of the main loop, we find the strongly connected components of the graph \( G \) and test if any bottom component (one that does not have any incoming edges) is a tie.

Notice that if both sides of the tie are nonempty, the roles of \( K \) and \( L \) in the algorithm are chosen arbitrarily; that is, there is nondeterminism in the algorithm because every such tie can be broken in one of two ways. Intuitively, all atoms in either set support atoms in the same set and oppose atoms in the other set, and thus either choice is reasonable. In general, these choices influence the outcome of the algorithm. For example, there are \( \Pi, A \), for which the algorithm may succeed in finding
a total model if it breaks ties in one way, but does not find a fixpoint if it breaks ties in a different way. When we say henceforth that the tie-breaking algorithm has a certain property, we implicitly mean that the property holds for all choices. If one of the two sets of the tie is empty, we could still have let the algorithm choose arbitrarily; the choice to make all the atoms false is more consistent with the minimalist philosophy of the well-founded, stable model and other previous approaches.

Note nevertheless that it can happen that both sides of a tie contain unfounded sets, that is, atoms that would be given value false by the well-founded semantics. Thus, the pure tie-breaking scheme as given above may in some cases produce a (partial) model that is not consistent with that of the well-founded semantics. As a simple example, consider the propositional program with two rules

\[ p \leftarrow p, \neg q, q \leftarrow q, \neg p. \]

The ground graph is a cycle with one side containing two positive arcs and the other containing two negative arcs. The pure algorithm will set one proposition to true and the other to false. However, \( \{p, q\} \) is an unfounded set, so the well-founded algorithm will set both to false. We can enforce consistency by combining the two approaches into one algorithm, which invokes the tie-breaking rule only when there is no unfounded set.

**Algorithm Well-Founded Tie-Breaking.**

\[ M := M(\Pi; A); G := G(\Pi, A); (M, G) := \text{close}(M, G); \]

while there is a change do:

- if \( C = \text{Atoms}[\text{close}(M, G)] \) is nonempty then
  - for each atom \( a \in C \) define \( M(a) := \text{false}; \)
  - \( (M, G) := \text{close}(M, G) \)
- else if there is a tie \( T \) in \( G \) with no incoming edges then
  - let \( (K, L) \) be the partition of \( T \) as in Lemma 1 with \( L \) nonempty;
  - for each atom \( a \in K \) set \( M(a) := \text{true}; \)
  - for each atom \( a \in K \) set \( M(a) := \text{false}; \)
  - \( (M, G) := \text{close}(M, G) \)

Both versions of the tie-breaking scheme are correct (for all choices) in the following sense.

**Lemma 2.** For every program \( \Pi \) and initial database \( A \), the partial model \( M^* \) computed by the (pure or well-founded) tie-breaking algorithm is consistent. Furthermore, if \( M^* \) is a total model, then it is a fixpoint.

**Proof.** We show that the computed partial model \( M^* \) is consistent, and furthermore, if an IDB atom \( a \) is true in \( M^* \), then it is supported, i.e., it is in \( A \) or there exists an instantiated rule \( r \) with head \( a \) such that all the literals in the body of \( r \) are true in \( M^* \). Clearly, \( M^* \) extends the initial model \( M_0(\Pi; A) \). Consider the procedure close \((M, G)\). If at some point a rule node \( r \) has no incoming edges, then this means that the predicate nodes corresponding to the literals in the body of \( r \) were previously deleted from the graph. Since \( r \) itself was not deleted, all the literals must be true. Thus, when an atom \( a \) is assigned value true then this is because it is the head of a rule \( r \) with true literals in its body.

If at some points an atom \( a \) is declared false because the corresponding predicate node does not have any incoming edges, then this means that all the rule nodes corresponding to rules with head \( a \) were previously deleted from the graph; hence each one of these rules has a false in its body.

Consider atoms that are assigned a truth value while breaking a tie \( T \) with partition \( K, L \). If a predicate node \( a \) is in \( K \), then it must have at least one arc from a rule node \( r \) because otherwise it would have been declared false and deleted by the procedure close \((M, G)\). Since \( T \) has no incoming arcs, \( r \) is in \( T \) and furthermore it must also belong to \( K \) because the arc \((r, a)\) is positive. All positive arcs into \( r \) come from nodes in \( K \) and all negative arcs come from nodes in \( L \). Thus, every literal in the body of \( r \) that was not previously deleted from the graph becomes true when the tie is broken. Furthermore, the literals corresponding to deleted nodes must be also true because otherwise \( r \) would have been deleted. Thus, all literals in the body of \( r \) are true.

If a predicate node \( a \) belongs to \( L \), then all its incoming arcs in the ground graph originate at rule nodes \( r \) which either were earlier deleted or belong also to \( L \). The former must have been deleted because they contain a false literal in their body. Each of the latter rule nodes \( r \) must have at least one incoming arc from a predicate node \( b \); either the arc is positive and \( b \) is in \( L \) or the arc is negative and \( b \) is in \( K \). In either case the corresponding literal in the body of \( r \) becomes false.

Finally, consider atoms that are assigned the value false because they belong to an unfounded set \( C \). In the subgraph of \( G_+ \) induced by \( C \) and the rule nodes that precede them, every node has at least one incoming edge. Thus, if \( a \) is an atom in \( C \), every rule \( r \) with head \( a \) contains in its body a positive atom \( b \) which also belongs to \( C \) and thus is assigned the value false.

A stronger claim can be made for the well-founded version of the algorithm.

**Lemma 3.** If the well-founded tie-breaking algorithm applied to a program \( \Pi \) and database instance \( A \) constructs a total model \( M^* \), then \( M^* \) is a stable model of \( \Pi \) and \( A \).

**Proof.** Let \( M^* \) be the partial model obtained from \( M^* \) by letting the true (IDB) atoms that are not in \( A \) be undefined. The procedure close \((M^*, G)\) will extend \( M^* \) to a consistent model, say \( M^+ \). Suppose that some atom is true in \( M^+ \) but not in \( M^* \). Let \( a \) be such an atom which was assigned value as early as possible in the construction of \( M^* \) by the well-founded tie-breaking algorithm. If this occurred during a call to the close \((M, G)\) procedure, then \( a \) is the head of a rule \( r \), all literals in the body of the rule received...
the value true earlier; hence they all have the value true in $M'$ and therefore $a$ should also be true in $M'$.

Suppose that the algorithm assigned the value true to $a$ while breaking a tie $T$ with partition $(K, L)$. Let $G$ be the graph at that point and let $M$ be the model; since the algorithm broke a tie, it could not find a nonempty unfounded set in $G$ with respect to $M$. Let $D$ be the set of the atoms of $K$ that are not true in $M'$; the set $D$ contains $a$ and, hence, is nonempty. Since $D$ is not the set of the atoms of $K$ that are not true in $M'$, the pure tie-breaking algorithm; i.e., this version may produce a fixpoint that is not a stable model. For instance, the pure tie-breaking algorithm will set one proposition to true and the other to false, which does not yield a stable model. The only stable model has both propositions false.

The conclusion of the previous lemma does not apply to the pure tie-breaking algorithm; i.e., this version may produce a fixpoint that is not a stable model. For instance, consider again the earlier example of the propositional program with rules $p \leftarrow p, \neg q$ and $q \leftarrow \neg p$. The pure algorithm will set one proposition to true and the other to false, which does not yield a stable model. The only stable model has both propositions false.

The converse of the lemma does not hold; that is, a program may have a stable model that cannot be produced by the well-founded (or pure) tie-breaking algorithm. Consider for example the propositional program with three rules $r_1: p_1 \leftarrow \neg p_2, \neg p_3; r_2: p_2 \leftarrow \neg p_1, \neg p_3; \text{and } r_3: p_3 \leftarrow \neg p_1, \neg p_2$. The ground graph $G$ consists of one strongly connected component containing the three proposition nodes $p_1, p_2, p_3$, and corresponding rule nodes $r_1, r_2, r_3$. The component is not a tie as it contains a cycle $r_1 \rightarrow p_1 \rightarrow r_2 \rightarrow p_2 \rightarrow r_3 \rightarrow p_3 \rightarrow r_1$ with three negative arcs. Furthermore, $G$ consists of three disjoint arcs $r_i \rightarrow p_i$, $i = 1, 2, 3$, and thus there is no nonempty unfounded set. Hence the well-founded tie-breaking algorithm will not assign a truth value to any proposition. However, there are three stable models, where each of them has one true and two false propositions; for example the model with $p_1 = \text{true}$ and $p_2 = p_3 = \text{false}$ is stable.

It is easy to see that both versions of the tie-breaking semantics extend the locally stratified semantics of [Pr]. A program is locally stratified if no strongly connected component of its ground graph contains any negative edges. Przymuński showed that every locally stratified program $II$ with initial database $D$ has a fixpoint, and in particular he defined a specific fixpoint called the perfect model which minimizes positive literals at lower levels. Notice that a strongly connected component with no negative edges is trivially a tie: one side $K$ is empty and the other side $L$ contains all the nodes. The tie-breaking algorithm (either version), when applied to a locally stratified program, will compute a fixpoint; in fact it is easy to see that it will compute the perfect model.

It is not hard to find examples of programs $II$ and databases $A$, where one version of tie-breaking succeeds in computing a fixpoint but not the other. However, as we shall see in the next section, in a structural sense, the well-founded tie-breaking algorithm subsumes all other methods.

We now state a simple sufficient condition for the tie-breaking interpreters to yield a total model under any choices. The only way that the algorithms will terminate prematurely failing to assign truth value to all the atoms is if at that point, the remaining graph does not have any bottom components that are ties. Suppose that the ground graph $G(II, A)$ has no cycles with an odd number of negative edges; that is, all strongly connected components of $G(II, A)$ are ties. Then, clearly the tie-breaking interpreter will produce a total model.

For a program $II$ define the program graph of $II$, $G(II)$, to be a graph with the predicate names as nodes, and a positive (respectively, negative) edge from node $P$ to node $Q$ if $P$ appears positively (resp., negatively) in the body of a rule with predicate $Q$ in its head. If the ground graph $G(II, A)$ contains a (possibly nonsimple) path from an atom of predicate $P$ to an atom of predicate $Q$, then the program graph $G(II)$ contains a corresponding path from node $P$ to node $Q$ that contains the same number of negative edges. Therefore, if the program graph does not contain a cycle with an odd number of negative edges then the same is true of the ground graph.

Kunen calls programs $II$ whose program graph $G(II)$ has this property, call-consistent; he proves that any such program has at least one fixpoint [Ku]. Furthermore, more recently Dung shows that any such program has at least one stable model [Du]. Gire calls these programs semi-strict, and he shows that for such programs the well-founded model is total if and only if there is a unique stable model (which is equal to the well-founded model) [Gi]. The proof techniques used in the above papers and in this section are similar. Note that Lemmas 2 and 3 imply that every “call-consistent” (or “semi-strict”) Datalog program has a fixpoint, and in fact a stable model. Observe also that the well-founded tie-breaking interpreter deviates from the well-founded interpreter if the latter gets stuck (i.e., the well-founded model is not total), in which case it breaks a tie arbitrarily in one of two ways, and both ways lead eventually to (different) stable models.

Summarizing our discussion and Lemmas 2 and 3, we have:

**Theorem.** If $G(II)$ has no cycle with an odd number of negative edges, then both the pure and the well-founded tie-breaking interpreter yield a total model (independently of
the initial database \( A \) and the choices) which is a fixpoint of \( \Pi \) and \( A \). Furthermore, the well-formed version yields a stable model.

4. STRUCTURAL TOTALITY

Our goal is to prove a strong converse of Theorem 1, stating that, whenever a program does not satisfy the odd cycle condition, then not only it has no tie-breaking fixpoint, but in fact it has no fixpoint whatsoever. In other words, to show that the tie-breaking semantics is the most general fixpoint semantics possible. The intuition is sound: A cycle with an odd number of negative edges encodes a contradiction that should rule out fixpoints. But there is a serious difficulty: The cycle may fail to function as an impediment to fixpoints because the variable names in the rules of the program fail to transfer the information necessary in order to arrive at the contradiction (recall the example of program (1)).

We want to rule out such counterexamples. In doing this, we seek more than mathematical elegance: Our goal here is to identify styles of programming with negation (say, generalizations of stratification) that lead to well-defined, total programs. And coincidental incompatibilities of variable names are clearly not the right concept to base our theory on. Let us call a program \( \Pi \) total (respectively, nonuniformly total) if it has a fixpoint for all initial database values \( A \) (respectively, with all IDBs empty). For each Datalog program \( \Pi \), we define its skeleton (or propositional form) to be \( \Pi \) with all parentheses, variables, and constants omitted. Finally, let us say that a program \( \Pi \) is structurally total (resp; structurally nonuniformly total) if all programs with the same skeleton as \( \Pi \) are total (resp. nonuniformly total).

**Theorem 2.** A program \( \Pi \) is structurally total if and only if \( G(\Pi) \) has no cycle with an odd number of negative edges.

**Proof.** The if direction follows immediately from Theorem 1. For the only if direction, let \( \Pi \) be a program such that \( G(\Pi) \) has a cycle \( C = (P_0, P_1, ..., P_k) \) with an odd number of negative edges. We shall construct an alphabetic variant \( \bar{\Pi} \) and an initial database \( A \) such that \( \bar{\Pi} \) has no fixpoint for \( A \). All predicates are unary in \( \bar{\Pi} \). Let \( a, b, c \) be three distinct constants. The initial database \( A \) contains \( Q(b) \) for all predicates \( Q \).

The program \( \bar{\Pi} \) is defined as follows. For every arc \( (P_i, P_{i+1}) \) of the cycle \( C \) (addition in the subscripts is modulo \( k + 1 \)), the skeleton of \( \bar{\Pi} \) contains a rule \( r_i \) of the form \( P_{i+1} \leftarrow (\neg) P_i \), where the \( P_i \) literal in the body is positive or negative depending on the sign of the arc. In \( \bar{\Pi} \) this rule becomes \( P_{i+1}(a) \leftarrow (\neg) P_i(a) \), where every other positive occurrence of any predicate \( Q \) in the body is replaced by \( Q(b) \) and every negative occurrence by \( \neg Q(c) \). In every other rule that does not “participate” in the odd cycle \( C \), positive occurrences of a predicate \( Q \) (either in the head or the body) are made \( Q(b) \) and negative occurrences are made \( \neg Q(c) \). This concludes the definition of \( \bar{\Pi} \).

Suppose that \( \bar{\Pi} \) has a fixpoint \( M \) for the initial database \( A \). Since \( Q(b) \) is true in \( A \) for all \( Q \), the rules that do not participate in the cycle do not play any role. Since \( Q(c) \) does not appear in the head of any rule, it must be false in the fixpoint \( M \). Hence, each rule \( r_i \) of the cycle simplifies into \( P_{i+1}(a) \leftarrow (\neg) P_i(a) \) since the rest of the literals in the body are true in \( M \). The fixpoint \( M \) induces a partition of the atoms \( P_i(a), i = 0, ..., k \), into two sets, True and False. Note that the atom \( P_{i+1}(a) \) does not appear in the head of any other rule besides \( r_i \). Therefore, if the arc \( (P_i, P_{i+1}) \) is positive (respectively, negative), then \( P_{i+1}(a) \) has the same (resp. opposite) truth value as \( P_i(a) \) in \( M \). This contradicts the fact that the cycle has an odd number of negative edges.

The program \( \bar{\Pi} \) above uses constants. The theorem can be shown also for constant-free programs, if we so desire. Let us predicates be ternary now. We use patterns of equalities in the arguments of the predicates in the rules to simulate the constants. Instead of the constant \( a \), use the triple of variable arguments \( (x, y, y) \); instead of \( b \), use \( (y, y, y) \); and instead of \( c \), use \( (x, x, y) \). Suppose the universe has two constants 1, 2, and let the initial database \( A \) contain all atoms of the form \( Q(d, d, d) \), where \( d = 1, 2 \). Any fixpoint must give value false to all atoms that have the first two arguments equal to each other and different from the third, because they do not unify with the head of any rule. Each rule \( r_i \) of the cycle instantiated with the values \( x = 1, y = 2 \) will simplify to \( P_{i+1}(1, 2, 2) \leftarrow (\neg) P_i(1, 2, 2) \). Thus, these rules and the atoms \( P_i(1, 2, 2) \) form a cycle with an odd number of negative edges, contradicting as above the existence of a fixpoint.

In the nonuniform case, the only if direction has the additional difficulty that we can only control the EDB relations. For example, some IDB relations may stay empty. We must take care of these first. Let \( \Pi \) be a program, and \( P \) a predicate symbol. An expansion of \( P \) in the skeleton of \( \Pi \) is a tree \( T \) whose nodes are labeled with predicate symbols \( Q \) or their negations, \( \neg Q \), such that: (1) the root is labelled \( P \) and (2) every nonleaf node \( v \) is labelled with some (positive) predicate \( Q \), and the skeleton of \( \Pi \) has a rule with head \( Q \) whose body consists of the labels of the children of \( v \). We call a predicate symbol \( P \) useful if it has an expansion \( T \) in the skeleton of the program, such that all the leaves of \( T \) are either negative predicates or EDB predicate symbols; all other predicates are useless. These notions can be related to the notions of useful and useless variables (nonterminals) of context-free grammars [HU], if one associates a context-free grammar with the skeleton of \( \Pi \) where positive IDB literals are treated as nonterminals and negative literals and EDB predicates are treated as terminals.
Eqivlently, the set of useless predicates is the largest set \( D \) if IDB predicates such that every rule whose head belongs to \( D \) contains in its body also a positive occurrence of some predicate of \( D \). This definition is related to the well-founded semantics as applied to the skeleton of \( \Pi \) in the following way. Regard the skeleton as a program \( \Pi_S \) in propositional variables, and consider an initial database \( \mathcal{A}_0 \) that consists of the propositions corresponding to the EDB relations, i.e., in the corresponding partial model \( \mathcal{M}_0(\mathcal{A}_0) \) the EDB propositions are true and the IDB are undefined. Then the largest unfounded set with respect to this partial model is precisely the set of propositions that correspond to the useless predicates.

Let \( \Pi' \) be the reduced Datalog program that results from \( \Pi \) if we omit all rules, where useless predicates appear positively, and omit all negative appearances of useless predicates in rules. This amounts to treating all useless predicates as empty. We have the following lemma.

**Lemma 4.** A program \( \Pi \) is structurally nonuniformly total if and only if the reduced program \( \Pi' \) is.

**Proof.** Suppose that \( \Pi' \) is structurally nonuniformly total. Let \( \tilde{\Pi} \) be a program with the same skeleton as \( \Pi \), and let \( \mathcal{A} \) be an initial database with all IDB predicates empty. Let \( \tilde{\Pi}' \) be the program obtained from \( \tilde{\Pi} \) as above by deleting all rules that contain positive appearances of useless predicates and by deleting all negative appearances of useless predicates in rules. Clearly, \( \tilde{\Pi}' \) has the same skeleton as \( \Pi' \). Let \( \mathcal{M}' \) be a fixpoint of \( \tilde{\Pi}' \) for the initial database \( \mathcal{A} \). The model \( \mathcal{M}' \) assigns value to all the literals of the useful predicates of \( \tilde{\Pi} \). Augment this to a model \( \mathcal{M} \) of \( \tilde{\Pi} \) by letting all the useless predicates be empty. Since every rule whose head is a useless predicate has in its body a positive occurrence of some useless predicate, it follows that \( \mathcal{M} \) is a fixpoint for \( \tilde{\Pi}, \mathcal{A} \).

Suppose that \( \Pi' \) is not structurally nonuniformly total. Let \( \tilde{\Pi}' \) be a program with the same skeleton as \( \Pi' \) and \( \mathcal{A} \) an initial database with all IDB predicates empty, for which there exists no fixpoint. Extend \( \tilde{\Pi}' \) to a program \( \hat{\Pi} \) that has as its skeleton the same \( \Pi' \) as \( \Pi' \) by letting the occurrences of useless predicates \( \mathcal{Q} \) at the heads of rules be \( \mathcal{Q}(a) \) for some constant \( a \) and by letting occurrences at the bodies be \( \mathcal{Q}(b) \) for a different constant \( b \), occurrences of useful predicates in the deleted rules are filled in arbitrarily. Suppose that \( \hat{\Pi}, \mathcal{A} \) has a fixpoint \( \mathcal{M} \). For every useless predicate \( \mathcal{Q} \), we must have \( \mathcal{Q}(b) \) be false in \( \mathcal{M} \) because \( \mathcal{Q}(b) \) does not appear in the head of any rule. Since all rules of \( \hat{\Pi} \) that are omitted from \( \hat{\Pi} \) have a false literal \( \mathcal{Q}(b) \) in their body, they do not contribute anything to the fixpoint. It follows that the values of the useful predicates in \( M \) form a fixpoint of \( \hat{\Pi}', \mathcal{A} \).

**Theorem 3.** A program \( \Pi \) is structurally nonuniformly total if and only if \( G(\Pi') \) has no cycle with an odd number of negative edges.

**Proof.** The **if** direction follows from Theorem 1 and Lemma 4.

For the **only if** direction, we are given a program \( \Pi \) such that \( G(\Pi') \) has a cycle \( C = (P_0, P_1, ..., P_k) \) with an odd number of negative edges. We must construct an alphabetic variant \( \hat{\Pi} \) and a set of values for the EDBs such that \( \hat{\Pi} \) has no fixpoint for the given set of EDB values.

The program \( \hat{\Pi} \) is defined as follows. Let \( a, b \) be two distinct constants. For every arc \( (P_i, P_{i+1}) \) of the cycle \( C \) (addition in the subscripts is again modulo \( k+1 \)), the skeleton of \( \Pi' \) contains a rule \( r_i \), of the form \( P_{i+1} \leftarrow \neg P_i(x), ... \), where the \( P_i \) literal in the body is positive or negative depending on the sign of the arc. In \( \hat{\Pi} \) this rule becomes \( P_{i+1}(a, x) \leftarrow P_i(a, x), ... \), if the arc is positive, and it becomes \( P_{i+1}(a, x) \leftarrow \neg P_i(x, a), ... \), if the arc is negative, where every other positive occurrence of any predicate \( \mathcal{Q} \) in the body is replaced by \( \mathcal{Q}(a, b) \) and every negative occurrence by \( \neg \mathcal{Q}(b, a) \). In every other rule that does not “participate” in the odd cycle \( C \), positive occurrences of a predicate \( \mathcal{Q} \) (either in the head or the body) are made \( \mathcal{Q}(a, b) \) and negative occurrences are made \( \neg \mathcal{Q}(b, a) \). This concludes the definition of \( \hat{\Pi} \). In the initial database \( \mathcal{A} \) all EDB relations are initialized to \( \{ (a, b) \} \), and all IDB relations to \( \emptyset \).

Suppose that \( \hat{\Pi} \) has a fixpoint \( \mathcal{M} \) for \( \mathcal{A} \). First we observe that \( \mathcal{Q}(b, a) \) must be false for very predicate \( \mathcal{Q} \), because if every rule has \( a \) as its first argument and hence does not unify with \( \mathcal{Q}(b, a) \). Next we shall show that \( \mathcal{Q}(a, b) \) must be true in \( \mathcal{M} \) for every useful predicate \( \mathcal{Q} \).

Consider the following iterative procedure for ordering the IDB predicates. In each iteration, we choose (if there is one) a predicate \( \mathcal{Q} \) that has not been chosen so far such that there is a rule \( r \) with \( \mathcal{Q} \) in its head, and all the positive literals in the body of \( r \) are either EDB predicates or predicates that have been already chosen. The procedure terminates if it cannot choose any new predicate with this property. Let \( \mathcal{D} \) be the set of remaining IDB predicates at the end. For every \( \mathcal{Q} \in \mathcal{D} \), every rule with head \( \mathcal{Q} \) contains in its body a predicate of \( \mathcal{D} \). Thus, \( \mathcal{D} \) contains useless predicates. In fact, it is easy to see that it contains all of them: For, suppose this is not the case, and consider the earliest useless predicate \( \mathcal{Q} \) that is chosen by the procedure. Then there is a rule \( r \) with \( \mathcal{Q} \) as its head such that all the positive literals in the body are either EDB predicates or IDB predicates that were already chosen and, hence, are useful; this contradicts the assumption that \( \mathcal{Q} \) is a useless predicate.

Therefore, we can order the useful IDB predicates as \( \mathcal{Q}_1, \mathcal{Q}_2, ..., \) so that for every \( j \) there is a rule \( r_j \) with head \( \mathcal{Q}_j \) in its head, whose body consists of negative literals, EDB literals, or positive literals involving only lower indexed predicates \( \mathcal{Q}_1, l < j \).

We show that \( \mathcal{Q}_j(a, b) \) is true in \( \mathcal{M} \) for every \( \mathcal{Q}_j \) by induction on \( j \). Suppose first that the above rule \( r_j \) is not one of the rules \( r_i \), of the cycle \( C \). Then the head of the rule is \( \mathcal{Q}_j(a, b) \).

TIE-BREAKING SEMANTICS 55
In the body of \( r_j \), the negative literals and the EDB literals are true by the construction of \( \bar{H} \) and \( A \), and the positive IDB literals are true by the induction hypothesis. Therefore, \( Q_j(a, b) \) must be also in true in \( M \). Suppose that \( r_j \) is one of the rules \( r_i \) of the cycle \( C \). If the corresponding arc is positive, then the instantiated rule is \( P_{i+1}(a, b) \leftarrow P_i(a, b), ..., \) and if the arc is negative then it is \( \neg P_{i+1}(a, b) \leftarrow \neg P_i(a, b), ... \). In either case, all literals in the body are true in \( M \) again, and hence the head must be also true.

Consider now the instantiation of the rules originating from the cycle \( C \) with \( x = a \). Since all IDB predicates that appear positively in the body are useful, the rules simplify to \( P_{i+1}(a, a) \leftarrow P_i(a, a) \) or \( P_{i+1}(a, a) \leftarrow \neg P_i(a, a) \) depending on the sign of the arc. These are the only rules of the program where the head unifies with these atoms. Therefore, the implication in these rules is actually an equivalence in the fixpoint. As in Theorem 2, this contradicts the fact that \( C \) has an odd number of negative arcs.

In this case too, we can construct a constant-free program that does not have a fixpoint by using predicates of higher arity, four. A rule \( r_j \) corresponding to an arc of the odd cycle \( C \) becomes \( P_{i+1}(x, y, y, z) \leftarrow P_i(x, y, y, z), ... \) or \( P_{i+1}(x, y, y, z) \leftarrow \neg P_i(x, y, y, z), ... \) depending on whether the arc is positive or negative. All other positive occurrences of a predicate \( Q \) in these or other rules, whether in the body or the head, become \( Q(x, z, z, z) \). All other negative occurrences of a predicate \( Q \) in the body of these or other rules become \( \neg Q(x, x, y, y) \). Suppose that the universe contains constants \( 1, 2 \); the initial database \( A \) contains \( Q(1, 2, 2, 2) \) for all EDB predicates \( Q \).

The head of all the rules has the second and third arguments equal. Therefore, any fixpoint must have \( Q(2, 1, 2, 2) \) false for all predicates \( Q \). Next, we claim that for every useful predicate \( Q_j \), a fixpoint must have \( Q_j(1, 2, 2, 2) \) true. This can be shown by an induction on the index \( j \) using similar arguments as before. For example, if the rule \( r_j \) corresponds to a negative arc of \( C \), then its instantiation becomes \( P_{i+1}(1, 2, 2, 2) \leftarrow \neg P_i(1, 2, 1, 2), ... \), where positive literals in the body of the form \( Q(1, 2, 2, 2) \) and negative literals are of the form \( \neg Q(1, 1, 2, 2) \); thus all the literals are true. Finally, consider the atoms \( P_i(1, 1, 1, 2) \) for the predicates \( P_i \) of the cycle \( C \). Since the fourth argument is not equal to the third, this atom unifies only with the head of the rule corresponding to the arc of the cycle that enters node \( P_i \). The instantiation of this rule simplifies to \( P_i(1, 1, 1, 2) \leftarrow P_{i-1}(1, 1, 1, 2), \) or \( P_i(1, 1, 1, 2) \leftarrow \neg P_{i-1}(1, 1, 1, 2) \), depending on whether the arc is positive or negative, because the other literals in the body are true.

It follows from Theorems 2 and 3 that structural totality is an easy property to check.

**Theorem 4.** Structural totality of Datalog programs can be checked in linear time, and is in NC. Structural non-uniform totality can be checked also in linear time, but is P-complete.

**Proof.** Given a program \( H \), we can construct its program graph \( G(H) \), find its strongly connected components, and test whether every component is a tie as described in the previous section. Each of these steps can be performed in linear time, and in NC, using standard techniques. Thus, we can test for structural totality in the uniform case in linear time and in NC.

In the nonuniform case, we have to determine first the useless predicates, transform the program \( H \) to the reduced program \( H' \), and test the graph \( G(H') \) for odd cycles as above. We can easily find the useful predicates (and thus also the useless) in linear time, by the procedure described in the proof of Theorem 3. However, it is P-complete to tell whether a predicate is useless.

We give now the P-completeness proof of structural non-uniform totality. The reduction is from the monotone circuit value problem. We are given a Boolean circuit \( B \) composed of \( \land \) and \( \lor \) gates, with \( n \) input bits and one output; we are also given an assignment \( x \) for the input bits and wish to determine whether the output \( B(x) = 0 \) or 1. We shall construct a program \( H \) which is structurally nonuniformly total if and only if \( B(x) = 0 \). We only need to specify the skeleton of \( H \). We have a predicate \( G_i \) for every gate and every input of the circuit and an additional predicate \( P \). If the \( i \)th input bit \( x_i \) is 1, then the corresponding predicate \( G_i \) is an EDB predicate; if \( x_i = 0 \) then \( G_i \) is an IDB predicate with one rule \( G_i \leftarrow G_j \); thus \( G_i \) is a useless predicate. A predicate \( G_i \) that corresponds to a \( \land \) gate is the head of exactly one rule whose body contains (positively) all the predicates corresponding to the inputs of the gate. In the case of a \( \lor \) gate, there are as many rules as inputs to the gate: each such rule has the predicate corresponding to the gate in its head and the predicate corresponding to the input in its body. Finally, we have a rule \( P \leftarrow \neg P, G_m \), where \( G_m \) is the predicate corresponding to the output gate of the circuit. It is easy to show inductively that a predicate \( G_i \) is useful if and only if the corresponding gate has value 1. It follows that the reduced program \( H' \) contains the odd cycle caused by the rule of \( P \) if and only if \( B(x) = 1 \).

From Theorems 1 and 2 if a program \( H \) is structurally total (in the uniform sense), then for any database \( A \), the tie-breaking algorithm (either version, with any arbitrary choices) will construct a fixpoint. In the nonuniform case, we first have to set the useless predicates to empty and remove them. Note that there may exist cycles with an odd number of negative edges involving the useless predicates.

The well-founded version of the tie-breaking algorithm will be able to identify at the beginning the useless predicates, set their atoms to false, and remove them.
When we have an arbitrary program $II$ and database $d$ it is possible that there exists a fixpoint (and specifically, even one constructible by the tie-breaking algorithm), but that there is none that extends the well-founded partial model. By contrast, in the structural case we have:

**Corollary 1.** Suppose that $II$ is structurally total (respectively, in the nonuniform sense), and that $A$ is a database (resp. with all the IDB predicates empty). Then $II, A$ has a fixpoint that extends the well-founded (possibly partial) model and which, furthermore, is computable in polynomial time. In particular, the well-founded tie-breaking algorithm computes such a fixpoint.

We used fixpoint semantics to define totality. The characterization in the structural setting would have been the same if we had used the stable model semantics instead.

**Corollary 2.** A program is structurally total (respectively, in the nonuniform sense) if and only if every program with the same skeleton has at least one stable model for every database $A$ (resp. with all the IDB predicates empty).

**Proof.** The if direction follows from the definition because a stable model is a fixpoint. The only if direction follows from Corollary 1 and the fact that any fixpoint computed by the well-founded tie-breaking algorithm is a stable model (Lemma 3).

We end this section with a characterization of programs that are total under the well-founded semantics (again, in a manner preserved under alphabetic variants). Surprisingly, it turns out that only stratified programs are: Let us call a that are total under the well-founded semantics (again, in a

**Theorem 5.** A program $II$ is structurally well-founded total (resp. in the nonuniform sense) if and only if it is stratified, i.e., $G(II)$ has no cycle with a negative edge (resp., the reduced program $II'$ is stratified).

**Proof Sketch.** For the if direction, it is known that the well-founded algorithm computes a total model when applied to stratified programs. In the nonuniform case, the atoms of the useless predicates form an unfounded set, so the well-founded algorithm will set them to false and proceed to compute a total model for the reduced program.

The only if direction can be shown using the same constructions as in Theorems 2 and 3, starting from a cycle $C = (P_n, P_{n-1}, ..., P_1)$ that contains a negative arc. Using similar arguments, one can show that the well-founded algorithm will determine the ground atoms that must be true or false in all fixpoints. As in the proof of Theorems 2 and 3, we will be left at the end with the rules of the cycle, simplified and instantiated in the form $P_{i+1}(\tau) \leftarrow \neg (\gamma) P_i(\tau)$, where $\tau$ is a ground tuple that does not unify with the head of any other rules. That is, these rules and the atoms $P_i(\tau)$ form a cycle in the ground graph $G$ that has no other incoming edges. If the cycle did not contain any negative arcs, then the atoms $P_i(\tau)$ would form an unfounded set and the well-founded algorithm would set them to false. However, since there is at least one negative arc, the atoms do not belong to any unfounded set; in forming the positive subgraph $G_+$ we would remove the negative arcs, breaking the cycle in the process, and if we applied then close($M, G_+$) we would remove these atoms. Therefore, the well-founded algorithm will not be able to assign a truth value t the atoms $P_i(\tau)$.  

A similar result holds for structural totality with respect to the unique stable model semantics. That is, a program $II$ has the property that every alphabetic variant has a unique stable model for every database $A$ if and only if $II$ is stratified (and the property holds for every $A$ with the IDB predicates empty iff $II'$ is stratified). The if direction is trivial. The only if direction can be seen as follows. Corollary 2 says that the graph $G(II)$ of a program $II$ with the above property cannot contain an odd cycle. By Gir’s result, such a program has a unique stable model iff it has a total well-founded model $[Gi]$. Theorem 5 implies then that $II$ must be stratified.

5. TOTALITY

Given a program $II$ we wish to test if it is total in the uniform or nonuniform sense, i.e., if it has a fixpoint for all initial database values $A$, resp. will all IDBs empty. This is not an easy problem. In the simple propositional case, the problem is decidable, but it is already NP-hard; more precisely, it is complete for the second level $P_2$ of the polynomial hierarchy.

**Proposition.** Given a propositional program, it is $P_2$-complete to determine whether it is total in the uniform or nonuniform sense.

**Proof.** In the propositional case, a database and a fixpoint are simply truth assignments to the propositions of the program. Thus, membership in $P_2$ follows directly from the definition of totality. For the hardness part we reduce from the following $P_2$-complete problem. Given a Boolean formula $F(x, y)$ in conjunctive normal form whose variables are partitioned into two parts, $x = x_1, ..., x_n$ and $y = y_1, ..., y_m$, determine whether it is the case that for all assignments to $x$, there is an assignment to $y$ that satisfies $F$. We give a construction that works for both uniform and nonuniform totality.

For every variable $x$, we have an EDB proposition, which we denote by $X_i$; for every variable $y$, we have an EDB proposition $Y_i$. We have two additional IDB propositions $p$ and $q$. For every clause $C_j$ of $F$, the program $II$ has a rule
with head $p$, whose body contains $\neg p$, $\neg q$ and the negations of the literals of $C_j$; i.e., the literal $x_0$, $\neg x_0$, $y$, or $\neg y$, is in the body iff the clause $C_j$ contains the complementary literal $\neg x_0$, $x_0$, $\neg y$, or $\neg y$, respectively. In addition the program has the following rules for every variable $y$:

1. $Y_i \leftarrow Y_i, \neg q$;
2. $q \leftarrow Y_i, q$.

Suppose that there is an assignment to the variables $x$ such that for all $y$, the formula $F(x, y)$ is false, i.e., at least one of the clauses is not satisfied. Consider the initial database which gives value to the EDB propositions $X_i$ according to this assignment and leaves the IDB propositions undefined.

Observe that in a fixpoint, for $q$ to be true, some $Y_i$ must be true. However, setting $Y_i$, to true requires $q$ to be false. Thus, we conclude that any fixpoint must set $q$ to false and will set some of the propositions $Y_i$ to true and the rest to false. This corresponds to a truth assignment to the variables $y_i$ of $F$, where $y_i$ is true iff $Y_i$ is true. This assignment to $y_i$ combined with the assignment to the variables $x_i$, violates one of the clauses. The corresponding rule of the program simplifies to $p \leftarrow \neg q$. It follows that $p$ cannot be set to either true or false, and there is no fixpoint.

Suppose that for every assignment to the variables $x$ there is an assignment to $y$ that satisfies the formula $F(x, y)$. Consider any initial database $A$ which gives value to the EDB propositions $X_i$ and possibly contains also some IDB propositions (i.e., sets them true). Give value to the undefined IDB propositions to construct a fixpoint as follows. First, suppose that all IDB propositions are undefined. Then, let $p$ and $q$ be false, and choose truth values for the propositions $Y_i$ according to the truth assignment to the variables $y_i$ that satisfies $F(x, y)$. Note that all rules with head $p$ have a false literal in their bodies because all clauses are satisfied.

Suppose that some IDB propositions are true in the initial database $A$. If $p$ is true in $A$ then let $q$ be false and give arbitrary values to the $Y_i$. Suppose that $p$ is undefined. If $q$ is true in $A$, then set $p$ and all undefined $Y_i$ to false; note that $q$ does not need to be supported by a rule since it is contained in the initial database. Also note that having $q = true$ disables all the rules with head $p$. If $q$ is undefined, but a $Y_i$ is true in $A$, then set $q$ true, and let again $p$ and all undefined $Y_i$ be false. It is easy to check that the model we described is a fixpoint, proving the theorem.

Starting the reduction from the ordinary satisfiability problem and omitting the propositions $X_i$ and $q$ shows that totality (in the uniform or nonuniform sense) with respect to the stable and the well-founded semantics is coNP-hard. In the case of the well-founded semantics, totality is actually in coNP. For the stable model semantics we do not know if it is in coNP.

In the general predicate case, the problem is undecidable. Checking nontotality is r.e.: guess a bad database $A$ and verify that there is no fixpoint.

**Theorem 6.** It is undecidable to test whether a given program is total in the uniform or nonuniform sense.

**Proof.** We describe a reduction from the halting problem for deterministic 2-counter machines. We first give the construction for the nonuniform case (all IDB predicates initialized to empty), and then we shall modify it for the uniform case.

Let $M$ be a 2-counter machine $M$. Assume that the states of the machine $M$ are numbered 0, 1, ..., $h$, where 0 is the starting state (with 0 in both counters) and $h$ is the halting state. The program will apply negation only to EDB predicates except for one rule. We have three binary IDB predicates $\text{STATE}(T, S)$, $\text{COUNT1}(T, C_1)$, $\text{COUNT2}(T, C_1)$ an IDB proposition $p$, and EDB predicates $\text{zero}(Z)$, $\text{succ}(X, Y)$, $\text{less}(X, Y)$. The tuples of the binary IDB predicates are supposed to encode the configurations of the machine, where the tuples $\text{STATE}(t, s)$, $\text{COUNT1}(t, c_1)$, $\text{COUNT2}(t, c_2)$ mean that at time $t$ the machine is in state $s$ with $c_1$ and $c_2$ in its two counters. The part of the Datalog program that simulates the moves of the machine is similar to a reduction of $\text{G + }$ for boundedness. The rules of the program are as follows:

** Initialization:**

$\text{STATE}(T, S) \leftarrow \text{zero}(T), \text{zero}(S)$,
$\text{COUNT1}(T, C_1) \leftarrow \text{zero}(T)\text{zero}(C_1)$
$\text{COUNT2}(T, C_2) \leftarrow \text{zero}(T)\text{zero}(C_2)$.

For a variable $X$ and a natural number $i$, we let $[X = i]$ abbreviate the conjunction $\text{zero}(A_0), \text{succ}(A_0, A_1), ..., \text{succ}(A_{i-1}, X)$, where the $A_j$’s are distinct variables that do not appear elsewhere in the rule. For every transition rule of $M$, the program $H$ has a corresponding set of three rules, one each for the $\text{STATE}$, $\text{COUNT1}$, and $\text{COUNT2}$ predicates. Suppose for example that when the machine is in state $s$, counter 1 is nonzero, and counter 2 is zero, it moves to state $s’$, does not change counter 1, and increments counter 2. The corresponding rule of $H$ for $\text{STATE}$ is as follows:

$\text{STATE}(T’, S’) \leftarrow \text{STATE}(T, S)$, $\text{COUNT1}(T, C_1)$,
$\text{COUNT2}(T, C_2)$, $\text{succ}(T, T’)$,
$\neg \text{zero}(C_1), \text{zero}(C_2)$, $[S = s]$, $[S’ = s’]$.

There are analogous rules for the $\text{COUNT1}$ and $\text{COUNT2}$ predicates corresponding to the transitions.

The possibly troublesome rule of $H$ is $p \leftarrow \neg p$, $\text{STATE}(T, S)$, $[S = h]$. Finally, we have the following rules which allow us to set $p$ to true in certain cases that the EDB
predicates zero and succ do not have their natural meaning in the initial database:

\[
\begin{align*}
(1a) & \quad p \leftarrow \text{succ}(X, Y), \neg \text{less}(X, Y); \\
(1b) & \quad p \leftarrow \text{succ}(X, Y), \text{less}(Y, Z), \neg \text{less}(X, Z); \\
(2) & \quad p \leftarrow \text{STATE}(T, S), \text{STATE}(T, S'), [S' = h], \text{less}(S, S').
\end{align*}
\]

We shall show that the program II is nonuniformly total if and only if the machine M halts.

Suppose that M halts, say in t steps. We let the universe contain 0, 1, ..., t (assuming without loss of generality that \( t > h \)). Let the initial database \( D \) assign the empty relation to the IDB predicates, and the natural meaning to the EDB predicates: zero contains 0, succ contains the pairs \((i - 1, i)\), and less the pairs \((i, j)\) with \( i < j \). We shall argue that there is no fixpoint.

We claim that in any fixpoint, the IDB predicates \text{STATE}, \text{COUNT1}, \text{COUNT2} must contain the tuples that correspond to the configurations that occur during the computation of the machine M and cannot contain any more tuples (recall, we do not insist on least fixed point). This can be shown by an induction on the first component \( t \) of a tuple in an IDB predicate. For \( t = 0 \), the tuple must be supported by the initialization rule and, thus, it is the tuple that corresponds to the initial configuration. For \( t > 0 \), the tuple must be obtained from a tuple with first component \( t - 1 \) using a transition rule; thus, the claim follows from the induction hypothesis.

The bodies of all rules of \( p \) are falsified except for the troublesome rule which becomes \( p \leftarrow \neg p \) because M halts. Therefore, there is no fixpoint.

Conversely, suppose that the machine M does not halt. Let \( D \) be any database, where the IDB predicates are empty. We compute the least fixed point for the three binary IDB predicates in the usual way (these rules do not use negation, except on the EDB predicate zero). If the conjunction \( \text{STATE}(T, S), [S = h] \) in the body of the troublesome rule is not true, then there is no problem. If the conjunction is true, then we shall show that one of the other rules can be used to deduce \( p \) true.

Suppose that \( \text{STATE}(t, s) \), and zero(a_0), succ(a_0, a_1), ..., succ(a_{n-1}, s) are true for some constants \( t, s, a_0, ..., a_{n-1} \) of the universe. Observe that every rule with head \( \text{STATE}(T, S) \) contains in the body either zero(T) or succ(T, T'). Since the least fixed point for the three binary IDB predicates does not contain any IDB facts and \( \text{STATE}(t, s) \) can be derived from \( D \) using the initialization and transition rules, it follows that there are elements \( b_0, b_1, ..., b_{j-1}, b_j = t \), such that \( D \) contains zero(b_0) and succ(b_i, b_{i+1}) for all \( i = 0, ..., j - 1 \).

We can have the rules of the program simulate the moves of the machine using the \( b_i \)s for the time and the counter components of the predicates and using the \( a_i \)s for the state component. Suppose that at time \( t \) the machine M is in state \( i \neq h \). We can infer then \( \text{STATE}(t, a_i) \). If \( p \) cannot be inferred from rules 1a and 1b, then the relation less contains the transitive closure of the relation succ; therefore, less(a_i, s) is true. It follows that rule 2 implies the proposition \( p \), and thus there is a fixpoint.

For the uniform case, we modify the program II into a program II' as follows. Add an IDB proposition \( q \). In the body of every rule of II we add the literal \( \neg q \). Furthermore, for every IDB predicate \( Q \) of II we include in II' the rule \( q \leftarrow Q(z) \), where \( z \) is a tuple of distinct variables. We claim that II is nonuniformly total if and only if II' is uniformly total.

For the \((if)\) part, let \( A \) be a database with all IDB predicates empty and suppose that II' has a fixpoint \( F \) for \( A \). For \( q \) to be true in the fixpoint, some \( Q(z) \) must be true this in turn requires \( q \) to be false because the literal \( \neg q \) occurs in the bodies of all the rules and \( A \) does not include any IDB facts. Therefore, \( q \) must be false in \( F \). Simplifying the new program II' by setting \( q \) to false yields the old program II. Thus, \( F \) yields a fixpoint for II, \( A \).

For the \((only if)\) direction, assume that II is nonuniformly total and consider an initial database \( D \) for II'. Suppose that \( A \) contains \( q \) or some IDB atom \( Q(z) \). Then we can form a fixpoint for II' by letting \( q \) be true, and setting to false all other IDB atoms that are undefined in \( A \). If \( A \) does not include any IDB atoms, then form a fixpoint by setting \( q = \text{false} \) combined with a fixpoint for II, \( A \).

**Corollary 3.** It is undecidable to determine whether a given program is nonuniformly total with respect to the stable model, the well-founded semantics, or the tie-breaking semantics.

**Proof.** Consider the above reduction in the nonuniform case. If the machine M halts then there is no fixpoint at all, and thus the program is not total with respect to any of the semantics. If M does not halt, then, as we argued in the proof of Theorem 6, for any database \( A \) with empty IDB predicates, there is a fixpoint that can be computed as the least fixed point of all the rules of the program except for the troublesome rule. As we explained in the proof of Theorem 6, we will never have to deal with the troublesome rule because either an atom in the body of that rule is false or the head is derived by another rule. The rest of the program, besides the troublesome rule, applies negation only to EDB predicates, and thus its least fixed point is consistent with all the semantics.

We do not know whether the same corollary applied in the uniform case.

6. CONCLUSIONS AND OPEN PROBLEMS

We addressed the problem of characterizing those Datalog programs with negation that are well structured, in
the sense that they are guaranteed to always have a fixpoint. We showed that if we want to take into account the patterns of variables and constants in the rules and their intricate interactions, then the problem is undecidable. However, if we ignore the patterns of the variables and only consider the basic structure of the program as determined by the predicate names, then the problem of characterizing well-structured programs in this sense is solvable in polynomial time. Furthermore, a fixpoint in this case can be always computed using an extension of the well-founded semantics by a tie-breaking algorithm.

The archetypical unstratifiable program that is structurally total is

\[ P(x) \leftarrow \neg Q(x); \quad Q(x) \leftarrow \neg P(x). \]

It has two fixpoints: One of \( P(x), Q(x) \) is true, and the other is false. It is debatable whether such programs are meaningful and useful; we do not think that the answer is obvious either way. There are many occasions where it is meaningful for the programmer to let the interpreter nondeterministically “break ties.” This would not be the first language that supports nondeterminism. And Theorem 4 says that the syntax of such programs is easy to check. But no matter what the reader thinks about the usefulness of the tie-breaking semantics, it turns out that they capture the class of programs whose structure guarantees the existence of a fixpoint, and in this sense they are the missing link between previously known fixpoint semantics and the ultimate limitations of this approach.

Structural totality may seem at first sight a very restricted notion, since a program fails to be structurally total if an alphabetic variant of tremendously high arity is not total, even though all reasonable alphabetic variants may be total. The proofs of Theorems 2 and 3, however, show that this is never the case: If a program is not structurally total, then there is a rather simple alphabetic variant with binary predicates (unary in the uniform case) which is not total. The same can be said about structural well-founded totality (Theorem 5). Also notice that the undecidability of totality (Theorem 6) also uses at most binary predicates. If all the EDB and IDB predicates are unary, then one can easily show that totality is decidable. It would be interesting to determine the status of totality if only the IDB predicates are unary.

REFERENCES


