NEGATION AS FAILURE USING TIGHT DERIVATIONS FOR GENERAL LOGIC PROGRAMS*

ALLEN VAN GELDER†

A general logic program is a set of rules that have both positive and negative subgoals. We define negation in general logic programs as finite failure, but we limit proof attempts to tight derivations, that is, derivations expressed by trees in which no node has an identical ancestor. Consequently, many goals that do not fail finitely in other formulations do fail finitely in ours. Thus the negation-as-failure rule is strengthened, but at the cost of more careful (and expensive) program execution. We define the tight tree semantics as a pair of interpretations: SS, interpreted as the success set, and FS, interpreted as the failure set. We show that general logic programs with both the bounded-term-size property and freedom from recursive negation are "categorical" under the tight tree semantics; that is, every atom in the Herbrand base of the program is either in SS or FS. Then we show that programs with these properties have an equivalent iterated-fixed-point semantics, which has been studied by other researchers.

1. INTRODUCTION

Negation as failure has been studied extensively as a means of extending the power of logic programming without taking on the burden of full-fledged non-Horn resolution. The negation-as-failure rule was introduced by Clark [5], and is closely related to the closed-world assumption of Reiter [27]. Rigorous treatments of the Horn-clause case, in which rules have only positive subgoals, but the query may contain negative subgoals, may be found in [3, 10]. General logic programs, in which

---

*This paper is a revised version of one that appears in Foundations of Deductive Databases and Logic Programming (J. Minker, ed.), Morgan Kaufmann Publishers, Los Altos, CA, 1988.
†Supported by NSF grant IST-84-12791 and a grant of IBM Corp.
Address correspondence to Professor Allen Van Gelder, Department of Computer Science, University of California, Santa Cruz, CA 95064.
Accepted 10 September 1987

THE JOURNAL OF LOGIC PROGRAMMING
655 Avenue of the Americas, New York, NY 10010

0743-1066/89/$3.50
rules may also have negative subgoals, have been studied in [12, 28, 13, 23, 11] and elsewhere (see [12, 28] for further bibliography).

We develop a semantics for general logic programs that is based upon the use of tight derivations. We say a derivation tree is tight if no node is identical to one of its own ancestors. Requiring tightness is a technique for avoiding redundant proofs (and nonterminating searches) while maintaining completeness that has been exploited in the theorem-proving community [17].

Many of the infinite recursions for which PROLOG is infamous can be avoided by requiring derivations to be tight. The necessary checking is generally not done, because of the overhead and because it is unnecessary in many programs. The usability of PROLOG would be considerably increased by the development of algorithms both to detect the danger of infinite recursion at "compile time" and to execute the program in a way that enforces tightness, yet is reasonably efficient. This is one of the research objectives of the Nail! project at Stanford University [21]. (Nail! stands for "Not another implementation of logic!") This paper takes a first step in that direction by defining an appropriate semantics for negation as failure using tight derivations.

Another problem with negation as failure is that some goals "slip through the cracks", in the sense that they neither succeed not fail finitely. We define two properties of logic programs that "close up" these cracks. The bounded-term-size property ensures that no tight derivation tree has an infinite path, and freedom from recursive negation ensures that no two goals can "deadlock", each waiting for the other to fail. One of our main results is that programs with both these properties are "completely evaluated" by our semantics, in the sense that every goal is in either the success set or the failure set. Furthermore, when the bounded-term-size property is supplemented by a known bounding function \( f(n) \), it is decidable whether any goal in the Herbrand base succeeds or fails.

For programs with the two key properties, the bounded-term-size property and freedom from recursive negation, our second main result is that these programs have an equivalent iterated-fixed-point semantics. The iterated-fixed-point semantics can be formulated as truth in a certain circumscription theory [14], and is thought by several researchers to be the "natural" interpretation for a general logic program, [1, 14, 21, 30]; we shall give some examples to support this view.

Preliminary versions of this work have been presented in conferences [37].

2. GENERAL LOGIC PROGRAMS AND SAFE NEGATION

In this section we introduce our notation and basic definitions, and describe the class of general logic programs that we shall be considering in this paper. We define safe negation and exclude programs with unsafe negation from further consideration.

Definition 2.1. A general logic program is a set of general rules, which may have both positive and negative subgoals. A general rule is written with its head, or conclusion, on the left, and its subgoals, if any, to the right of the symbol " \( \leftarrow \) ", which may be read "if". For example,

\[ p(X) \leftarrow a(X), \text{not} \ b(X). \]

is a general rule in which \( p(X) \) is the head, \( a(X) \) is a positive subgoal, and \( b(X) \)
is a negative subgoal. We follow the PROLOG convention of capitalizing variables, but not other symbols.

A rule with no subgoals, or an instance of such a rule, is called a fact; the "←" is omitted when writing it. A Horn rule is one with no negative subgoals, and a Horn logic program is one with only Horn rules.

Further, we require a logic program to have a finite alphabet of constants, function symbols, and predicate symbols, and to have a set of variables that is indexed by the natural numbers.

We shall be considering atoms in the Herbrand base and ground rules whose variables have been instantiated to elements of the Herbrand universe.

Definition 2.2. The Herbrand instantiation of a general logic program P is the set of ground rules P_H obtained by substituting terms in the Herbrand universe for variables in every possible way. An instantiated logic program P ↓ is any subset of P_H. Whereas “uninstantiated” logic programs are assumed to be a finite set of rules, instantiated logic programs may well be infinite.

For simplicity in our discussion, we exclude equality (=) from the language. The extension to include equality is straightforward, following the methods presented in [35], where E-unification replaces unification. For example, we say a ground atom appears in a set if it unifies with an atom in the set. Of course, in our discussion ground terms or atoms unify if and only if they are identical. In an extension that includes equality, ground terms or atoms would be said to appear in a set if they E-unify with a member of the set.

2.1. Safe and Unsafe Negation

Unsafe negation is essentially complementation with respect to an ill-defined domain. Informally, unsafe negation occurs when we try to solve a negative subgoal containing free variables; we discuss some examples before giving a formal definition. Safe negation is similar to various other concepts, including “flounder-free” query evaluation [5,28], “sound negation” [23], “safe formulas” in relational calculus [31,38], and “domain-independent formulas” [6,30].

One trivial source of unsafe negation can be eliminated syntactically. To avoid ambiguity in the quantification of a variable, we shall require that any variable that appears in a negative subgoal must also appear in the head of the rule or in a positive subgoal, thereby confirming that the variable is universally quantified at the scope of the entire rule. This involves no loss of generality, as rules may always be rewritten using the technique of projection to ensure that no variable appears only in negative subgoals.

Example 2.1. Consider a program P_1 consisting of the following rather natural-appearing rule:

\[
bachelor(X) \leftarrow male(X), \text{not married}(X,Y).
\]

If Y is considered universally quantified at the scope of the rule, and this quantifier is pushed down, it becomes a universal quantifier immediately above the atom
married$(X, Y)$. When PROLOG undertakes the goal \textit{married}(X, Y), however, Y is necessarily free, and so is existentially quantified, in effect. To avoid this source of confusion we shall insist that the rule be rewritten, e.g., as the pair of rules $P'_1$:

\begin{align*}
\text{married}(_1(X)) & \leftarrow \text{married}(X, Y). \\
\text{bachelor}(_1(X)) & \leftarrow \text{male}(X), \text{not} \text{married}(_1(X)).
\end{align*}

A more troublesome problem with unsafe negation arises when some subgoals can succeed without binding their arguments to ground terms.

\textit{Example 2.2.} Let $P_2$ be the program

\begin{align*}
p(X) & \leftarrow a(X), d(X, Y). \\
d(X, Y) & \leftarrow s(X, X), s(Y, Y), \text{not} s(X, Y). \\
s(U, U). \\
a(1).
\end{align*}

The subgoals $s(X, X)$ and $s(Y, Y)$ in the rule for $d$ are not necessary, but are included to emphasize the point that a program may be unsafe for negation even if every variable in a negative subgoal also appears in a positive subgoal. In the rule-based semantics, $SS$ contains only $a(1)$ and $s(1, 1)$, while $d(1, 1)$ and $p(1)$ are in $FS$. However, adding the apparently unrelated fact $b(2)$ to the program means that now $d(1, 2)$ is in $SS$, and so $p(1)$ switches from $FS$ to $SS$. Such bizarre behavior is clearly undesirable in a practical system.

A detailed treatment of this problem is beyond the scope of this paper. The main idea is that in the example above the Herbrand universe is not a sufficiently large domain of interpretation. (See [19] for related discussion.) Operationally, we want the negative subgoal \textit{not} $d(1, Y)$ in the above example to succeed unless $d(1, Y)$ can “succeed without binding Y”. In general the problem is harder, and complex representations of goals are needed, along the lines introduced in [16]. However, to avoid these issues, we shall deal only with logic programs that are safe for negation in the remainder of this paper. To define this term, we first need to define top-down-positive derivations.

\textit{Definition 2.3.} For our purposes, a \textit{top-down-positive derivation} is a list $C_0, \ldots, C_n$, defined recursively:

1. Each $C_i$ is a conjunctive clause, called a line of the derivation, made up of positive and negative subgoals.
2. The single line $C_0$ is a top-down-positive derivation, where $C_0$ is the top-level goal or conjunction of goals.
3. Let $C_0, \ldots, C_k$ be a top-down-positive derivation, and let a new clause $C_{k+1}$ be formed from $C_k$ as follows:
   \begin{itemize}
   \item[(a)] Select a positive subgoal $p$ in $C_k$. Following Lloyd [15], we call the (computable) procedure $\phi$ for selecting the subgoal the \textit{computation rule}. 
   \end{itemize}
(b) Using a most general unifier $\theta$, unify $p$ with the head of some rule $R$, whose variables have been renamed to those of least index that do not occur in $C_0, \ldots, C_k$.

(c) The $C_{k+1}$ consists of $C_k\theta$ with $p\theta$ removed and the subgoals of $R\theta$ added.

If $C_{k+1}$ is so formed, then $C_0\theta, \ldots, C_k\theta, C_{k+1}$ is also a top-down-positive derivation.

A positive-complete derivation is a top-down-positive derivation in which the last clause has no positive subgoals.

Note that a proof procedure does not simply extend a derivation, but due to the action of $\theta$, both modifies and extends it. Informally, a proof procedure based on $\phi$ generates a coherent sequence of derivations.

Definition 2.4. A general logic program is safe for negation if every top-down-positive derivation has the property that any variable in a negative subgoal is also in some positive subgoal of the same derived clause or in the top-level clause.

The essential property of programs that are safe for negation is that whenever the top-level clause of a positive-complete derivation is variable-free (ground), then so is the last clause, which contains only negative subgoals (or is empty). We see that this is not the case in $P_2$ above, because there is a positive-complete derivation that begins with $p(1)$ and ends with $\text{not} \ s(1, Y)$. However, experience with practical logic programs suggests that they usually are safe for negation.

One approach is to require that every variable in a rule appear in a positive subgoal [28,1,30]. It is clear that this ensures that the program is safe for negation, but it eliminates many “workhorse” routines of logic programming, such as

$$\text{member}(X, X, R).$$

$$\text{member}(X, A, R) \leftarrow \text{member}(X, R).$$

A less restrictive syntactic condition that ensures safety for negation would be useful, but is not easy to find. An algorithm that identifies a large class of safe-for-negation programs will appear in a future report and will be incorporated into the Nail! system.

3. FREEDOM FROM RECURSIVE NEGATION

The concept of freedom from recursive negation, defined below, is an important one for logic programs. The same concept was developed independently in [1, 24], where it was called “stratified” and “layered”, respectively. Other researchers adopted the term “stratified” [14,26], but we prefer our own more descriptive terminology.

The definition of recursive negation is based on the concept of a dependence graph of the program. The nodes of this graph are the predicates of the logic program. Whenever predicate $p$ is in the head of a rule and predicate $q$ appears as a

---

1 Such a query is called flounder-free [15,28].
(positive or negative) subgoal, we put an arc \( p \rightarrow q \) into the graph. This done, we identify the strongly connected components (SCCs) of the graph, which are the maximal sets of nodes that can all reach each other. The *reduced graph* of the dependence graph is obtained by combining all nodes in an SCC into a single node, and eliminating arcs among them. Hence the reduced graph is acyclic. SCC nodes with no out-arcs are considered leaves. Finally we define the *rank* of an SCC as its height in the reduced graph, with leaves having height 0.

*Example 3.1.* Consider the following program, which is a variation of one that is discussed in Example 5.1.

\[
p(X, Y) \leftarrow b(X, Y).
p(X, Y) \leftarrow b(X, U), p(U, Y).
e(X, Y) \leftarrow g(X, Y), \text{not } p(X, Y).
a(X, Y) \leftarrow e(X, Y).
a(X, Y) \leftarrow e(X, U), a(U, Y).
\]

These rules together with facts for \( b \) and \( g \). This program's dependence graph is

\[
\begin{tikzpicture}
  \node (a) at (0,0) {a};
  \node (e) at (1,0) {e};
  \node (p) at (2,0) {p};
  \node (b) at (3,0) {b};
  \node (g) at (1,-1) {g};

  \draw[->] (a) -- (e);
  \draw[->] (e) -- (p);
  \draw[->] (p) -- (b);
  \draw[->] (g) -- (e);
\end{tikzpicture}
\]

Eliminating the self-arcs gives the reduced graph in this case, from which we see that the ranks of \( b \) and \( g \) are 0, the rank of \( p \) is 1, the rank of \( e \) is 2, and the rank of \( a \) is 3.

In a loose sense the ranks of SCCs correspond to levels of abstraction. In keeping with this view, we do not expect users to define less abstract items in terms of more abstract ones. Thus we do not view a rule like \( p \leftarrow \text{not } q \) as a proper mechanism for deducing \( q \) when \( p \) is false; this is the essential difference between logic programs with negative subgoals and "pure" non-Horn clauses. However, if \( p \) and \( q \) are in the same SCC, then they are both at the same level of abstraction, and we feel that the user's intentions are then unclear. This motivates the following definition.

*Definition 3.1.* We say a general logic program contains *recursive negation* if some rule has a negative subgoal in the same SCC as its head. A program in which this does not occur is said to be *free of recursive negation*.

4. **RULE-BASED NEGATION-AS-FAILURE SEMANTICS**

In this section we describe informally the semantics of general logic programs with negation as failure interpreted as the "traditional" finite failure [5,3,10,15]. The corresponding proof mechanism is frequently called *SLDNF resolution*.

For a given general logic program \( P \), we shall be concerned with three kinds of sets: *success sets* \( (SS_k) \), *failure sets* \( (FS_k) \), and *remaining-rules sets* \( (RR_k) \). To get started, we define \( SS_0 = FS_0 = \emptyset \), and define \( RR_0 = PH \), the Herbrand instantiation.
of $P$. For any ordinal $k$, suppose we have three sets, $SS_k$, $FS_k$, and $RR_k$:

$SS_k$ is a set of atoms that are considered true.

$FS_k$ is a set of atoms that are considered false.

$RR_k$ is an instantiated logic program.

We define the triple $(SS_{k+1}, FS_{k+1}, RR_{k+1})$ in terms of $(SS_k, FS_k, RR_k)$ by means of a transformation $\Phi$, which we now describe. First create $SS_{k+1}$ and $FS_{k+1}$ by adding atoms to $SS_k$ and $FS_k$, if possible, as follows:

For each rule in $RR_k$ with no subgoals, add the head of that rule to $SS_{k+1}$.

For each atom in the Herbrand base that does not unify with the head of any rule in $RR_k$, add that atom to $FS_{k+1}$.

Now create $RR_{k+1}$ by modifying $RR_k$ as follows:

Initialize $RR_{k+1}$ to $RR_k$.

Delete each rule in $RR_{k+1}$ that has a positive subgoal in $FS_{k+1}$ or has a negative subgoal in $SS_{k+1}$; these rules can never succeed.

In each remaining rule in $RR_{k+1}$, delete all positive subgoals that appear in $SS_{k+1}$ and delete all negative subgoals that appear in $FS_{k+1}$; these subgoals are considered proved and the rules "shrink" accordingly.

This completes the description of the transformation $\Phi(SS_k, FS_k, RR_k)$. Figure 2 in Section 6 illustrates the operation of $\Phi$.

For each limit ordinal $\alpha$, we define

$$SS_\alpha = \bigcup_{\beta < \alpha} SS_\beta, \quad FS_\alpha = \bigcup_{\beta < \alpha} FS_\beta, \quad RR_\alpha = \bigcap_{\beta < \alpha} RR_\beta.$$  

Let $\Omega$ be the least nonconstructive ordinal, and define

$$SS = \bigcup_{\beta < \Omega} SS_\beta, \quad FS = \bigcup_{\beta < \Omega} FS_\beta.$$  

Also, we say an atom in the Herbrand universe is unclassified if it is in neither $SS$ nor $FS$.

**Definition 4.1.** The rule-based negation-as-failure semantics (rule-based semantics for short) for a general logic program is the pair of sets $(SS, FS)$ that is derived from $SS_0$, $FS_0$, $RR_0$, and the transformation $\Phi$, as described above. Also, the "meaning of the logic program" with respect to the rule-based semantics is that atoms in $SS$ are true, atoms in $FS$ are false, and atoms in neither $SS$ nor $FS$ are unclassified.

This formulation is appealing for its uniformity and simplicity. Nevertheless, it can lead to some counterintuitive results, as shown in the next section.

---

2 The semantics of general logic programs requires transfinite ordinals, $0, 1, 2, 3, \ldots, \omega, \omega + 1, \omega + 2, \omega + 3, \ldots, 2\omega, 2\omega + 1, \ldots$. A successor ordinal is one that has a greatest ordinal less than itself, such as $1, 2, \omega + 1$, etc. Limit ordinals lack this property, such as $\omega, 2\omega$, etc.
When the logic program \( P \) consists of just Horn rules, and the query, or top-level goal, is allowed to be a conjunction of subgoals, some of which may be negative, then the rule-based semantics is "sound and complete" in a certain sense. To make this precise, we need the notion of the Clark completion of \( P \), \( \text{comp}(P) \). This is given formally in [5, 3, 10]; briefly, each rule is rewritten into an equivalent one such that each argument in the head is a distinct variable; then all rules with the same head (up to variants) are rewritten as one "superrule" by disjunctively connecting their bodies; finally all "ifs" are replaced by "iff's". For example, \( \text{comp}(\text{member}) \) (following Definition 2.4) is

\[
\text{member}(X, Y) \iff \exists R (Y = X \land R \land \exists A (Y = A \land R \land \text{member}(X, R)))
\]

**Theorem 4.1.** Let \( P \) be a Horn program, and let \( q \) be an atom in its Herbrand base. Let \( SS \) and \( FS \) be the success and failure sets, respectively, of \( P \), as defined above for the rule-based semantics.

Then, \( q \in SS \) if and only if \( \text{comp}(P) \) logically implies \( q \) [3].

Also, \( q \in FS \) if and only if \( \text{comp}(P) \) logically implies \( \neg q \) [10].

However, completeness does not extend to general programs, as shown in the next section by Example 5.2. That is, when \( P \) contains rules with negative subgoals, then \( \text{comp}(P) \) (or even \( P \) itself) may logically imply \( q \), but \( q \notin SS \).

5. COUNTERINTUITIVE MEANINGS OF RULE-BASED SEMANTICS

In this section we discuss several ways in which rule-based semantics may give counterintuitive meanings. We identify two categories of problems as failure to fail and indefinite case. Failure to fail leads into the principal work of this paper. Indefinite case is related to failure to fail.

5.1. Failure to Fail

There are certain programs where it is obvious to a person that certain facts are unprovable, but the rule-based semantics does not put them into \( FS \). In many cases they are related to PROLOG's well-known "left recursion loops".

**Example 5.1.** Suppose we have a directed graph with two kinds of arcs, "bad" arcs (represented by \( b \)) and "good" arcs (represented by \( g \)). We want to define the following relations on the graph:

- \( p(X, Y) \) holds when there is a "poor" path (a sequence of "bad" arcs) from \( X \) to \( Y \).
- \( e(X, Y) \) holds when there is an "excellent" path (a sequence of "good" arcs) from \( X \) to \( Y \).
- \( a(X, Y) \) (our top-level goal) holds when there is an "excellent" path and no "poor" path from \( X \) to \( Y \).

The foregoing definitions are apparently expressed by the program \( P_3 \), containing the
following rules:

\[
\begin{align*}
    p(X,Y) & \leftarrow b(X,Y), \\
    p(X,Y) & \leftarrow b(X,U), p(U,Y), \\
    e(X,Y) & \leftarrow g(X,Y), \\
    e(X,Y) & \leftarrow g(X,U), e(U,Y). \\
    a(X,Y) & \leftarrow e(X,Y), \text{not } p(U,Y).
\end{align*}
\]

Now suppose that the facts about \( b \) and \( g \) arcs are

\[
\begin{align*}
    b(1,2) & \quad g(2,3) \\
    b(2,1) & \quad g(3,2) \\
    b(3,4) & \\
    b(4,3)
\end{align*}
\]

as suggested by Figure 1. Intuitively, we expect \( a(2,3) \) and \( a(3,2) \) to be true and \( a(2,4) \) and \( a(3,1) \) (among others) to be false. The rule-based semantics with this program will duly compute \( SS \) that includes

\[
\{ p(1,1), p(1,2), p(2,1), p(2,2), \\
    p(3,3), p(3,4), p(4,3), p(4,4), \\
    e(2,2), e(2,3), e(3,2), e(3,3) \},
\]

but it will leave all of the abovementioned atoms for \( a \) unclassified. The problem is that atoms like \( p(1,3), p(2,3), e(2,4), e(3,4), \) etc. cannot be put into \( FS_\alpha \) for any ordinal \( \alpha \). This is because of loops such as the one expressed by

\[
\begin{align*}
    p(1,3) & \leftarrow b(1,2), p(2,3). \\
    p(2,3) & \leftarrow b(2,1), p(1,3).
\end{align*}
\]

Consequently, \( a(2,3) \) and \( a(3,2) \) are never put into \( SS \), because \( p(2,3) \) and \( p(3,2) \) never enter \( FS \). Similarly, \( a(2,4) \) never fails because \( e(2,4) \) never fails, and \( p(2,4) \) of course never succeeds.

5.2. Indefinite Case

In Example 5.1 the \( SS \) produced by the rule-based semantics was not a model, but did represent the intersection of all models. Here we show that \( SS \) very well may not be even that large. That is, there may be facts that are in all models, but are not

---

*By model we shall always mean Herbrand model.*
put into SS by the rule-based semantics. Thus the completeness of Theorem 4.1 does not extend to general programs.

**Example 5.2.** Let $P_4$ be a program with the rules

$$p(X) \leftarrow a(X), c(X).$$
$$p(X) \leftarrow \text{not } a(X), d(X).$$
$$a(X) \leftarrow b(X).$$
$$b(X) \leftarrow a(X).$$

together with the facts

$$c(1), \quad c(2), \quad d(2), \quad d(3).$$

We observe that $p(2)$ is in all models. However, the rule-based semantics leaves $a$, and hence $p$, completely unclassified. The problem is that one of the cases \{a(2), \neg a(2)\} must hold but the semantics is not powerful enough to use this information.

5.3. **Unintended Models**

Additional problems of counterintuitive consequences in default logics, which have a mechanism much like the negation-as-failure rule, are studied by Hanks and McDermott [7,8]. They argue that unintended minimal models, coupled with the requirement that a conclusion be true in all minimal models, make the reasoning power of such systems very weak.

**Example 5.3.** Referring to Example 5.1, consider the minimal models of $\text{comp}(P_3)$. The (presumably) intended model contains $a(2,3)$, and neither $p(2,3)$ nor $p(1,3)$. But there is another minimal model, which contains $p(2,3)$ and $p(1,3)$ and not $a(2,3)$. Thus none of these atoms is either in all or absent from all minimal models; hence they are neither true nor false by that standard.

Examples like this have led researchers to question the value of defining “true” to mean “in all models of $\text{comp}(P)$”, and instead to define “true” to mean “in a preferred model” [1,14,21,26,30]. In the remainder of this paper we shall describe an alternative proof system and show how it is connected to the “preferred model”.

6. **TREE-ORIENTED SEMANTICS FOR NEGATION AS FAILURE**

In this section we consider semantics in which remaining-rule sets $(RR_k)$ are replaced by sets of “remaining proof trees”. First we define a *simple tree semantics* that is equivalent to the rule semantics of Section 4 in the sense that it produces the same $SS_k$ and $FS_k$. Then we propose a modification called the *tight tree semantics* that provides a strengthened form of finite failure.
6.1. Simple Tree Semantics

The simple tree negation-as-failure semantics is defined analogously to rule-based semantics with sets of remaining NF-trees \((RT_k)\) replacing the sets of remaining rules \((RR_k)\).

**Definition 6.1.** Given an instantiated general logic program \(P \downarrow\), a negation-as-failure derivation tree (NF-tree for short) is a (possibly infinite) tree whose nodes are one of:

- a positive atom in the Herbrand base, which may have children;
- a negated atom in the Herbrand base, which must be a leaf;
- the constant \textit{failed}, which is considered to be neither an atom nor a negated atom for our purposes, and must be a leaf.

For each internal node, either it has a \textit{failed} child or there is some instantiated rule such that:

The internal node unifies with the head of the instantiated rule.

The children of the internal node can be placed in a one-to-one correspondence with matching subgoals of the rule (including the positive or negative polarity).

An \textit{active} NF-tree is one with no \textit{failed} nodes. A \textit{complete} NF-tree is an active one in which all of the positive leaves are facts in \(P \downarrow\). An active NF-tree is said to be \textit{incomplete at depth} \(d\) if some positive leaf at depth \(d\) from the root is not a fact in \(P \downarrow\).

Note that a complete NF-tree may be infinite, and may have no leaves.

**Definition 6.2.** An atom \(p\) is said to be \textit{NF-derivable} with respect to sets \(SS_k\) and \(FS_k\) and instantiated logic program \(RR_k\) if there is a finite NF-tree based on \(RR_k\) with \(p\) as root such that each leaf of the tree is either a positive atom in \(SS_k\) or the negation of an atom in \(FS_k\). In this case we say that the tree \textit{NF-derives} \(p\) with respect to \(SS_k, FS_k,\) and \(RR_k\).

As we did for the rule-based semantics, we define \(SS_0 = FS_0 = \emptyset\), and define \(RR_0 = PH\), the Herbrand instantiation of \(P\). Then \(RT_0\) is defined to be the set of all NF-trees based on \(RR_0\), as described in Definition 6.1.

For any ordinal \(k\), suppose we have the triple \((SS_k, FS_k, RT_k)\). We define the triple \((SS_{k+1}, FS_{k+1}, RR_{k+1})\) in terms of \((SS_k, FS_k, RT_k)\) by means of a transformation \(\Phi\), which we now describe. The close correspondence between this transformation and the \(\Phi\) defined for rule-based semantics is illustrated in Figure 2. First, create \(SS_{k+1}\) and \(FS_{k+1}\) by adding atoms to \(SS_k\) and \(FS_k\) as follows:

For each tree in \(RT_k\) consisting of a single node, add that atom to \(SS_{k+1}\).

For each atom in the Herbrand base that unifies with the root of no tree in \(RT_k\), add that atom to \(FS_{k+1}\).
Now create $RT_{k+1}$ by modifying $RT_k$ as follows:

Initialize $RT_{k+1}$ to $RT_k$.

In each tree in $RT_{k+1}$, for each internal node, if it has a child containing $\text{failed}$, then replace that node and its subtree by the constant $\text{failed}$. (Do this just once; do not further propagate a $\text{failed}$ that was created during this update step.) Trees that reduce to the single $\text{failed}$ node have failed finitely.

In each tree in $RT_{k+1}$, for each leaf, if it contains either a positive atom that appears in $FS_k$ or a negated atom that appears in $SS_k$, then replace that leaf by the constant $\text{failed}$.

In each tree in $RT_{k+1}$, delete all leaves (other than the root itself) that contain positive atoms that appear in $SS_k$, and delete all negated atoms (necessarily leaves) that appear in $FS_k$; these nodes are considered proved and the trees "shrink" accordingly.

This completes the description of the transformation $\Psi(SS_k, FS_k, RT_k)$. We note that $SS_k$ and $FS_k$ are monotonically increasing under $\Psi$. 

\begin{figure}
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
$k$ & $SS_k$ & $FS_k$ & $RR_k$ & $RT_k$ \\
\hline
0 & $\emptyset$ & $\emptyset$ & $p \leftarrow a, b.$ & \begin{tikzpicture}
  \node (p) at (0,0) {p};
  \node (a) at (-1,-1) {a};
  \node (b) at (1,-1) {b};
  \node (c) at (0,-2) {c};
  \draw (p) -- (a);
  \draw (p) -- (b);
  \draw (p) -- (c);
\end{tikzpicture} \\
\hline
1 & $c$ & $d$ & $e$ & \begin{tikzpicture}
  \node (p) at (0,0) {p};
  \node (a) at (-1,-1) {a};
  \node (b) at (0,-2) {b};
  \node (c) at (1,-1) {c};
  \node (d) at (0,-2) {d};
  \draw (p) -- (a);
  \draw (p) -- (b);
  \draw (p) -- (c);
  \draw (d) -- (b);
\end{tikzpicture} \\
\hline
2 & $c$ & $b$ & $d$ & $e$ & $a$ & $b$ & $c$ & \begin{tikzpicture}
  \node (p) at (0,0) {p};
  \node (a) at (-1,-1) {a};
  \node (b) at (0,-2) {b};
  \node (c) at (1,-1) {c};
  \node (d) at (0,-2) {d};
  \node (e) at (1,-2) {e};
  \draw (p) -- (a);
  \draw (p) -- (b);
  \draw (p) -- (c);
  \draw (d) -- (b);
  \draw (e) -- (c);
\end{tikzpicture} \\
\hline
3 & $c$ & $b$ & $d$ & $e$ & $a$ & $p$ & \begin{tikzpicture}
  \node (p) at (0,0) {p};
  \node (a) at (-1,-1) {a};
  \node (b) at (0,-2) {b};
  \node (c) at (1,-1) {c};
  \node (d) at (0,-2) {d};
  \node (e) at (1,-2) {e};
  \node (f) at (0,-3) {f};
  \draw (p) -- (a);
  \draw (p) -- (b);
  \draw (p) -- (c);
  \draw (d) -- (b);
  \draw (e) -- (c);
\end{tikzpicture} \\
\hline
\end{tabular}
\caption{Correspondence between rule-based semantics and simple-tree semantics ($\perp = \text{failed}$).}
\end{figure}
For each limit ordinal \( \alpha \), we define

\[
SS_\alpha = \bigcup_{\beta < \alpha} SS_\beta, \quad FS_\alpha = \bigcup_{\beta < \alpha} FS_\beta, \quad RT_\alpha = \bigcap_{\beta < \alpha} RT_\beta.
\]

Let \( \Omega \) be the least nonconstructive ordinal, and define

\[
SS = \bigcup_{\beta < \Omega} SS_\beta, \quad FS = \bigcup_{\beta < \Omega} FS_\beta.
\]

**Definition 6.3.** The *simple tree negation-as-failure semantics* (simple tree semantics for short) for a general logic program is the pair of sets \((SS, FS)\) derived from \(SS_0, FS_0, RT_0\), and the transformation \(\Psi\), as described above. Also, the “meaning of the logic program” with respect to the simple tree semantics is that atoms in \(SS\) are true, atoms in \(FS\) are false, and atoms in neither \(SS\) nor \(FS\) are unclassified.

Note that the treatment of the constant *failed* is designed to maintain correspondence with rule-based semantics (see Figure 2). The same values of \(SS\) and \(FS\) would be obtained if trees were completely failed as soon as one leaf failed, but the intermediate values of \(SS_\alpha\) and \(FS_\alpha\) would differ, in general.

**Lemma 6.1.** Given a general logic program, for all ordinals \( \alpha < \Omega \):

- The sets \(SS_\alpha\) and \(FS_\alpha\) in the rule-based semantics are the same as \(SS_\alpha\) and \(FS_\alpha\) in the simple tree semantics.
- Every subtree of a tree in \(RT_\alpha\) is also a tree in \(RT_\alpha\).
- Every tree in \(RT_\alpha\) is an NF-tree based on \(RR_\alpha\).
- For every rule in \(RR_\alpha\), there is a tree in \(RT_\alpha\) such that its root unifies with the head, the root has no failed child, and the children of the root unify with the subgoals of the rule. (Possibly failed nodes occur at lower levels.)

**Proof.** Use induction on \(\alpha\) and apply the definitions of \(\Phi\) and \(\Psi\). \(\square\)

**Theorem 6.2.** Given a general logic program, the sets \(SS\) and \(FS\) in the rule-based semantics are the same as \(SS\) and \(FS\) in the simple tree semantics.

**Proof.** Immediate from Lemma 6.1. \( \square \)

### 6.2. Tight Tree Semantics

In this section we show that the tight tree semantics avoids the counterintuitive “failure-to-fail” and “indefinite-case” results mentioned in Section 5 for a large class of logic programs.

**Definition 6.4.** A tight negation-as-failure derivation tree (tight NF-tree for short) is an NF-tree such that no node has an identical sector. Since ancestors are necessarily internal nodes, this condition only needs to be checked for nodes that are positive atoms.
Our strengthened form of finite failure is based on the use of tight NF-trees. Suppose we are trying to prove \( p \). Recall that Herbrand's theorem essentially guarantees that if a set of clauses (including \( \neg p \)) is inconsistent, then some finite subset is inconsistent; this finite subset can be used to produce a finite derivation of \( p \). Restricting attention to tight NF-trees is motivated by the following lemma.

**Lemma 6.3.** For a given instantiated logic program \( P \downarrow \), an atom is NF-derived with respect to sets \( SS_k \) and \( FS_k \) if and only if it is NF-derived by a tight NF-tree with respect to those sets.

**Proof.** Let \( t \) be a minimum (in node count) NF-tree that NF-derives atom \( p \). Suppose node \( v \) contains positive atom \( a \) and has an ancestor \( v' \) that also contains \( a \). Create a smaller NF-tree \( t' \) by replacing the subtree of \( t \) rooted at \( v' \) with the subtree rooted at \( v \). Clearly \( t' \) NF-derives \( p \), contradicting the minimality of \( t \). Therefore \( t \) must be tight. \( \Box \)

**Definition 6.5.** The tight-tree negation-as-failure semantics (tight-tree semantics for short) has \( \Psi \) defined as for simple tree semantics (Section 6.1). However, instead of starting at \((\emptyset, \emptyset, RT_0)\), the tight-tree semantics starts at \((\emptyset, \emptyset, TT_0)\), where \( TT_0 \) is defined to be the subset of \( RT_0 \) consisting of tight NF-trees. Thereafter,

\[
(SS_{k+1}, FS_{k+1}, TT_{k+1}) = \Psi(SS_k, FS_k, TT_k).
\]

Definitions for limit ordinals and \( SS \) and \( FS \) are as before.

**Example 6.1.** Recall the program \( P_3 \) from Example 5.1, which contained the rules

\[
\begin{align*}
p(X,Y) & \leftarrow b(X,Y). \\
p(X,Y) & \leftarrow b(X,U), p(U,Y). \\
e(X,Y) & \leftarrow g(X,Y). \\
e(X,Y) & \leftarrow g(X,U), e(U,Y). \\
a(X,Y) & \leftarrow e(X,Y), \text{not} \ p(U,Y).
\end{align*}
\]

and the facts about \( b \) and \( g \):

\[
\begin{align*}
b(1,2) & & g(2,3) \\
b(2,1) & & g(3,2) \\
b(3,4) & \\
b(4,3)
\end{align*}
\]

By considering only tight NF-trees, the looping problem disappears, and atoms like \( p(1,3) \) and \( p(2,3) \) can be put into \( FS_k \). For example, \( p(1,3) \) can reduce to \( b(1,2) \) and \( p(2,3) \), but then \( p(2,3) \) cannot reduce to \( b(2,1) \) and \( p(1,3) \), because tightness is violated. The tight-tree semantics with this program will therefore compute the more expected results that \( SS \) also includes \{ \( a(2,3), a(3,2) \) \} and \( FS \) includes \{ \( a(2,4), a(3,1) \) \}. 


Example 6.2. Recall that $P_4$ in Example 5.2 is the program with rules

\[
p(X) \leftarrow a(X), c(X).
\]

\[
p(X) \leftarrow \text{not} \ a(X), d(X).
\]

\[
a(X) \leftarrow b(X).
\]

\[
b(X) \leftarrow a(X).
\]

together with the facts

\[
c(1), \ c(2), \ d(2), \ d(3).
\]

The rule-based semantics did not put $p(2)$ into $SS$, although it is in all models, because it could not classify $a(2)$. The tight-tree semantics prohibits the infinite nontight tree $a(2) \leftarrow b(2) \leftarrow a(2) \leftarrow \cdots$, puts $a(2)$ into $FS_1$, and then puts $p(2)$ into $SS_2$. But observe that, by the same token, the tight-tree semantics puts $p(3)$ into $SS$, although it is not in all models. This illustrates the fact that the tight-tree semantics is associated with a “preferred” model.

6.3. The Bounded-Term-Size Property

Practical programs frequently have the bounded-term-size property, defined below, because the natural tendency in problem solving is to reduce larger objects to smaller ones. However, an important area for further investigation is the search for weaker properties that guarantee the same favorable characteristics as those enjoyed by programs with the bounded-term-size property. (Recent progress in this direction was announced in [29].) Another problem is that the bounded-term-size property is not constructively defined.

Definition 6.6. The size of a term is defined recursively:

The size of a variable is its index; recall that variables are indexed by the natural numbers.

The size of a constant is 1; recall that the set of constants is finite.

The size of a compound term is 1 plus the sum of the sizes of its arguments.

Definition 6.7. We say a general logic program has the bounded-term-size property if it is safe for negation and there is a function $f(n)$ and a (computable) computation rule $\phi$ such that whenever the top-level goal has no argument whose term size exceeds $n$, then no subgoal in any top-down-positive derivation based on $\phi$ (Definition 2.3) has an argument whose term size exceeds $f(n)$, whether the derivation is successful or not.

Example 6.3. Consider the three-argument reverse program $P_5$, in which it is intended that the first argument contain the unreversed part of a list, the second argument contain the already reversed part, and the third argument contain the complete reversed list. We assume lists are built with the binary functor “.” and that
it is defined as an infix operator.

\[ \text{rev}(\text{nil}, R, R). \]

\[ \text{rev}(\text{A} . \text{U}, P, R) ← \text{rev}(U, A . P, R). \]

We see that this program has the bounded-term-size property with \( f(n) = 2n \). Even though the goal

\[ ? \leftarrow \text{rev}(4 . 5 . 6 . \text{nil}, 3 . 2 . 1 . \text{nil}, \text{nil}) \]

eventually fails, we have to consider how large the terms can grow in the meantime. In this case the largest term size starts at 7 and grows to 13.

Example 6.4. Consider the transitive closure program \( P_6 \), in which \( e \) is given by a finite set of ground atoms, which do not concern us except that we denote the number of such facts by \(|e|\) and assume that all arguments of \( e \)-atoms are constants (term size 1):

\[ p(X, Y) ← e(X, Y). \]

\[ p(X, Y) ← e(X, U), p(U, Y). \]

We see that this program has the bounded-term-size property with \( f(n) = O(|e|) \). If the same \( e \) atom occurs twice in a derivation, then some \( p \)-atom also occurs twice, and one occurrence must be an ancestor of the other.

Now consider the transitive closure variant \( P_7 \), in which \( e \) is as before:

\[ p(X, Y) ← e(X, Y). \]

\[ p(X, Y) ← p(X, U), p(U, Y). \]

This program does not have the bounded-term-size property. For any computation rule \( \phi \), starting from goal \( p(a, b) \), it is possible to build derivations always using the second rule, hence only containing \( p \) atoms. It doesn’t matter which \( p \) atom \( \phi \) selects; a new variable will be introduced, so a derivation with arbitrarily many variables can be created. This example emphasizes two points: (1) variants of an ancestor do not violate tightness, and (2) \( \phi \) only selects the subgoal to be expanded, not the rule to apply.

It should be pointed out that it is undecidable in general whether a given program has the bounded-term-size property, for otherwise it would be possible to decide the halting problem.

Lemma 6.4. Let \( P \) be a general logic program that has the bounded-term-size property, with bounding function \( f(n) \). Then there is a function \( g(n) \) such that every tight NF-tree whose root contains terms of size bounded by \( n \) is finite and complete, or is incomplete at depth no greater than \( g(n) \) (Definition 6.1).

Proof. For our computation rule, we adopt “oldest first”, also called “breadth first”, that is, \( \phi \) selects a positive subgoal that was introduced earliest in the derivation.

Suppose ground atom \( r \) is the root of an NF-tree and \( n \) is a bound on the size of terms in \( r \). From the given NF-tree and \( \phi \) we construct a sequence of top-down-positive derivations \( D_k \) and mappings \( \mu_k \) from positive literals in \( D_k \) to nodes in the
NEGATION AS FAILURE

NF-tree, as follows:

(1) Begin with \( r \), the root of the NF-tree, as \( C_0 \), and say that \( \mu_0 \) maps the atom \( r \) in \( C_0 \) to the root \( r \). Call this one-line derivation \( D_0 \).

(2) Assume that the derivation \( D_k = (C_0, \ldots, C_k) \) and the mapping \( \mu_k \) have been defined, and that our computation rule \( \phi \) selects subgoal \( p \) in \( C_k \). Let \( \mu_k(p) \) be node \( v \) in the NF-tree. There are three cases:

(a) Line \( C_k \) has no positive subgoals, so no \( p \) exists. Then the NF-tree is complete and finite.

(b) Node \( v \) has no children and \( p \) is not an instance of any fact of \( P \). Then the NF-tree is incomplete at the depth of \( v \).

(c) A certain rule \( R \) produced the children of \( v \), if any. We extend \( D_k \) using the same \( R \) and most general unifier \( \theta \), producing a new derivation \( D_{k+1} = (C_0\theta, \ldots, C_k\theta, C_{k+1}) \). The new mapping \( \mu_{k+1} \) is defined so that the new subgoals that replaced \( p\theta \) map to the children of \( v \) in the NF-tree in a one-one fashion; other subgoals of \( C_{k+1} \) inherit their mappings from \( \mu_k \), as do subgoals in earlier lines.

Call a node in the NF-tree selected if some selected subgoal in the derivation sequence maps to it.

By hypothesis there is some \( f(n) \) that bounds the sizes of all terms in this sequence of derivations. The number of distinct variables in any one derivation is also limited to \( f(n) \), since a variable's size is given by its index. It follows that there is a function \( g(n) \) such that \( D_k \) exist only for \( k < g(n) \). [For each \( n \) there are finitely many choices for \( r \), so we can choose \( g(n) \) large enough for all of them.]

Thus case (a) or (b) above applies at a depth no greater than \( g(n) \). □

7. CATEGORICAL PROGRAMS UNDER THE TIGHT-TREE SEMANTICS

The concept of freedom from recursive negation, which was defined in Section 3, combined with the bounded-term-size property, leads to our first main result. In this section we show that programs that are free of recursive negation and have the bounded-term-size property are “categorical” under the tight-tree semantics, whereas programs with recursive negation are not necessarily categorical.

Definition 7.1. A program is said to be categorical with respect to a given semantics if every atom in the Herbrand universe is in either \( SS \) or \( FS \), and \( SS \) is a minimal model.

Theorem 7.1. Let \( P \) be a general logic program that is free of recursive negation and has the bounded-term-size property. Then the tight-tree semantics for \( P \) partitions its Herbrand base into \( SS \) and \( FS \); that is, every atom in the Herbrand base is classified. In addition, \( SS \) is a minimal (not necessarily minimum) model for \( P \).

PROOF. We partition the Herbrand base \( H \) according to the SCCs of the dependence graph into \( H_1, \ldots, H_m \), where the SCCs are labeled 1, \ldots, \( m \). Denote the rank of SCC \( u \) by \( rank(u) \). Let \( P^{(k)} \) denote the program consisting of the rules of \( P \).
whose heads have rank no greater than \( k \), and let

\[
SS^{(k)} = SS \cap \left( \bigcup_{\text{rank}(u) \leq k} H_u \right)
\]

with a corresponding definition for \( FS^{(k)} \). The critical observation is that the contents of \( SS^{(j)} \) and \( FS^{(j)} \), where \( j < k \) are not influenced by atoms in \( SS \cap H_v \) or \( FS \cap H_v \), where \( \text{rank}(v) = k \). We show by induction on \( k \), the rank, that

1. For all SCCs \( v \) with \( \text{rank}(v) = k \), \( SS \cap H_v \) and \( FS \cap H_v \) partition \( H_v \);
2. \( SS^{(k)} = SS^{(k+1)} \) and \( FS^{(k)} = FS^{(k+1)} \);
3. \( SS^{(k)} \) is a minimal model of \( P^{(k)} \).

For the base case, \( k = 0 \), (1) follows from Lemma 6.4 and the fact that \( P^{(0)} \) is a Horn-clause program, so that “NF-derivable” coincides with “has a resolution proof”. (To apply the lemma we note also that if every NF-tree with root \( r \) is incomplete at depth no greater than \( d \), then \( r \in FS^{(d+1)}_k \).) In this case \( SS^{(0)} \) is known to be the minimum model of \( P^{(0)} \) [34], establishing (2) and (3).

For the inductive step, let \( k > 0 \) and assume the inductive hypothesis for ranks less than \( k \). Let \( \alpha \) be an ordinal of the form \( k \omega + h \), where \( h \) is a natural number. Let \( v \) be any SCC with \( \text{rank}(v) = k \). By freedom from recursive negation we know that if some rule whose head is in the SCC \( v \) has a negative subgoal \( \text{not} \, q \), then \( q \) is in an SCC whose rank is \( j < k \). But \( SS^{(j)} = SS^{(j)}_\alpha \) and \( FS^{(j)} = FS^{(j)}_\alpha \) for the \( \alpha \) under consideration. Therefore, all such negative subgoals are classified in “step” \( k \omega + 1 \); i.e., they are either removed from the trees or replaced by failed. For the remaining \( \alpha \) in the range \( k \omega < \alpha < k \omega + \omega \) the tight-tree semantics behaves exactly as if the rules for SCCs at rank \( k \) were all Horn clauses. Therefore, (1), (2), and (3) follow by the same arguments used for rank 0. The only difference is that the model \( SS^{(k)} \) is minimal, but not necessarily minimum. That is, removing any atoms of rank \( j > 0 \) causes it to cease being a model, but then adding additional atoms of rank lower than \( j \) might restore the set to being a model that is not a superset of \( SS^{(k)} \) (cf. Example 5.3). \( \square \)

Example 7.1. To see that freedom from recursive negation (or some other condition in addition to bounded term sizes) is needed to prove the previous theorem, consider the program \( P_5 \):

\[
p \leftarrow \text{not} \, q.
\]

\[
q \leftarrow \text{not} \, p.
\]

We see that \( p \) and \( q \) are “deadlocked”, each waiting for the other to either succeed or fail. Thus neither \( p \) nor \( q \) is classified in the tight-tree semantics.

Corollary 7.2. Let \( P \) be a general logic program that is free of recursive negation and has the bounded-term-size property (under computation rule \( \phi \)). In addition, let the bounding function \( f(n) \) be known. Let \( SS \) and \( FS \) be based on the tight-tree semantics operating on \( P \). Then for any ground atom \( p \), it is decidable whether \( p \in SS \).
**Proof.** Use induction on the rank of \( p \). Perform a *tight* top-down proof search from top-level goal \( p \), using \( \phi \) as computation rule and using resolution in the manner of PROLOG on positive subgoals. Whenever variables are introduced, systematically try all substitutions of terms up to the allowable size \( f(n) \), where \( n \) is the size of the largest term in \( p \). (We didn’t say this was *efficient.*) By safety for negation, any variable in a negative subgoal also appears in a positive subgoal of the same clause (since the top-level clause was ground—see Definition 2.4), so \( f(n) \) is always sufficient. Because of the bound on term size and because we prohibit any subgoal that has an identical ancestor, the depth of this part of the proof search (i.e., expanding positive subgoals) is finite. If the rank of \( p \) is 0, we are done; otherwise, when a negative subgoal, say \( \neg q \), is encountered, by freedom from recursive negation, \( q \) has lower rank. By Theorem 7.1, \( q \) is in either \( SS \) or \( FS \), and by the inductive hypothesis, it is decidable which. Thus the proof search for \( p \) either succeeds or fails finitely. \( \square \)

It is hard to imagine a realistic situation in which we know that a bounding function \( f(n) \) exists, but we cannot exhibit any such function. One obvious case when \( f(n) \) is known is the function-free case. A less obvious, but at least partially analyzable, case is when function symbols are present, but the rules are such that arbitrarily large recursive term structures cannot be created. This is discussed further in the following section on practical methods.

When recursive terms can develop in the program, then analysis of \( f(n) \) is much more difficult, and only sketchy results are available [22,23,33]. These papers identify a few cases in which it can be proved that some argument’s term size definitely decreases around the recursive loop. Example 6.3 illustrated one such tractable case.

Recently, the preceding corollary was strengthened by Seki and Itoh [29]. They describe an algorithm that does not require knowledge of \( f(n) \) and also works for a weaker definition of "bounded-term-size property".

### 7.1. Iterated-Least-Fixed-Point Semantics

An alternative semantics for general logic programs that are free of recursive negation, and one that is closer to certain implementations, is based on iterated least fixed points. For example, when the Herbrand base is finite, as it is in the function-free case, then the fixed points are also finite, and can be computed by bottom-up methods. Our second main result is that the iterated-least-fixed-point semantics, which we describe in this section, and the tight-tree semantics are equivalent on general logic programs that have the bounded-term-size property and are free of recursive negation.

As is well known [34,2,3], a Horn-clause program \( P \) has a natural semantics expressed as the least fixed point of a *modus ponens* operator \( T_P \). Essentially, we define \( SS_0 = \emptyset \), and whenever \( SS_k \) contains all the subgoals of an instantiated rule, then the operator \( T_P \) puts the head of the instantiated rule into \( SS_{k+1} \). Then the success set \( SS \) of \( P \) is defined to be \( SS_w \), which is the least fixed point of \( T_P \). The rule-based semantics in Section 4 are in fact a generalization of this approach, in which the pair \((SS,FS)\) is defined to be the least fixed point of a transformation.
that operates on both \(SS_k\) and \(FS_k\) simultaneously. We shall consider a different generalization.

**Definition 7.2.** The operator \(U_p(I)\), where \(I\) is a set of atoms, is defined as follows:

- \(U_p(I)\) contains \(I\).
- For any instantiated rule in \(P\), if \(I\) contains all of the rule's positive subgoals and none of the rule's negative subgoals, then \(U_p(I)\) contains the head of the rule.
- \(U_p(I)\) contains no other atoms.

Integer powers (i.e., iterated applications) of \(U_p\) are denoted by \(U_p^k\), and we define

\[
U_p^\omega(I) = \bigcup_{k<\omega} U_p^k(I).
\]

For a given general logic program \(P\), we find its strongly connected components and their ranks as described in Section 3. Assume the program is found to be free of recursive negation. As before, let \(P(j)\) consist of the rules of \(P\) whose heads have rank \(j\). Then the iterated-fixed-point semantics of \(P\) is defined using the operators \(U_{P(j)}\):

- For rank 0,
  \[
  SS^{(0)} = U_{P^{(0)}}^\omega(\emptyset)
  \]
  Recall that \(P^{(0)}\) consists of Horn clauses. Therefore, \(SS^{(0)}\) is just the least fixed point of \(T_{P^{(0)}}\).
- For ranks \(j > 0\),
  \[
  SS^{(j)} = U_{P^{(j)}}^\omega(SS^{(j-1)})
  \]

To justify the least-fixed-point nomenclature for ranks \(j > 0\), we may define a modified program \(P^{(j)+}\) consisting of the atoms of \(SS^{(j-1)}\), regarded as facts, together with the rules of \(P^{(j)}\). Let \(D_j\) be the set consisting of the atoms whose arguments are in the Herbrand universe and whose predicates have rank \(j\), together with the atoms in \(SS^{(j-1)}\). Then

\[
SS^{(j)} = U_{P^{(j)+}}^\omega(\emptyset).
\]

All of the atoms added to \(SS^{(j)}\) are in \(D_j\) and, because \(P\) is free of recursive negation, none of these atoms can ever unify with a negative subgoal of a rule in \(P^{(j)+}\). This establishes the monotonicity of \(U_{P^{(j)+}}\) within \(D_j\), and thus \(SS^{(j)}\) is the least fixed point of \(U_{P^{(j)+}}\) restricted to that set.

**Theorem 7.3.** Let \(P\) be a general logic program that is free of recursive negation and has the bounded-term-size property. Then the iterated-fixed-point semantics of \(P\) and the tight-tree semantics of \(P\) are equivalent.

**Proof.** Use induction on rank. At rank 0 we have only Horn clauses. It is well known that the least fixed point of \(T_{P^{(0)}}\), which is the same as the least fixed point of \(U_{P^{(0)}}\), is the minimum model of \(P\), and that every atom in the minimum model has a
finite derivation tree [34, 3]. By Lemma 6.3, every such atom also has a tight NF-tree. Thus $SS^{(0)}$ coincides in both semantics.

For rank $j > 0$ assume that $SS^{(i)}$ coincides in both semantics for ranks $i < j$. By Theorem 7.1, every atom of rank $i < j$ that is not in $SS^{(i)}$ is in $FS^{(i)}$ in the tight-tree semantics. Now an easy induction on $k > 0$ shows that atom $p$ is added to $SS_{j}^{(i)}$ in the iterated-fixed-point semantics if and only if (based on $P^{(i)}$ as the program) there is a complete tight NF-tree of height at most $k$ that derives $p$ in the tight-tree semantics. It follows that $SS^{(j)}$ coincides in both semantics, completing the induction on $j$. □

Przymusinski has studied a weaker condition than freedom from recursive negation, called "local stratification" [26], in connection with the iterated-fixed-point semantics.

8. PRACTICAL ALGORITHMS

The decidability proof in Corollary 7.2 called for the algorithm to "try all possible terms up to size $f(n)$" at certain points. Practical algorithms cannot afford to "try all possible terms", of course. In many cases of practical interest, there is an order of goal reductions that allows all variables to be bound through unification, without invoking blind guessing. This is a research area in its own right.

The function-free case is an important special case that has received considerable study. In this case the finiteness of the Herbrand universe can be exploited. This case is doubly important because it includes many of the well-known examples where top-down derivations do not terminate. (Transitive closures, as in Example 5.1, are typical.) It has been argued [4] that top-down methods cannot be "patched up" to fix these problems, that essentially different algorithms are needed. Various proposals to handle the function-free case efficiently have appeared [20, 40, 9, 25, 36]. Their common themes are to introduce a bottom-up (forward reasoning) component into the evaluation procedure, while attempting to evaluate only relevant portions of the Herbrand base. Although these studies do not consider negative subgoals, many of their results carry over without trouble when the program is free of recursive negation.

8.1. Programs with Nonrecursive Term Structure

As soon as a program contains any functor of positive arity, the Herbrand universe becomes infinite. However, it is still possible that all tight NF-trees are finite. Essentially, for a tight NF-tree to be infinite, it must be possible for the program to generate larger and larger terms during a top-down-positive derivation. For example, a rule like

$$ht(X, Y) \leftarrow ht(s(X), Y).$$

does just that. One way to rule out this possibility is establish that the program cannot produce recursive terms at all, in either the top-down or the bottom-up

---

4 Actually, the interpreter described in [1] does so in the function-free case.
direction. In this case there is a finite "reachable" subset of the Herbrand universe. The success set of the program under any reasonable semantics will lie within this finite subset, so the methods developed for the function-free case can normally be applied.

We shall sketch the description of a sufficient condition that resolution using most general unifiers cannot lead to arbitrarily large recursive term structures. Clearly it is sufficient to analyze each SCC separately for this purpose. In the discussion below, "functor" means a function symbol of positive arity; a constant is not regarded as a functor here. The idea is to look at the most general unifiers associated with each possible match of a positive subgoal to a rule head, look at the compound terms that occur in the program as predicate arguments, and build a directed functor precedence graph whose nodes are functors and variables (rules have disjoint sets of variables). For each substitution in some most general unifier, put an arc from the variable to each functor of the substitution term; put arcs both ways if two variables unify. For each term in the program, put an arc from each functor to each variable within its term. If no functor is in a cycle now, recursive terms cannot be built, at least not through resolution with most general unifiers, and \( f(n) \) can be computed.

**Example 8.1.** Suppose the only nontrivial rules are

\[
p(f(X), g(X)) \leftarrow q(X, X).
\]

\[
q(U, V) \leftarrow p(h(U), V).
\]

Then the functor precedence graph is

```
   h
   v
```

```
f   g
```

```
U   X   V
```

Since no functors are in cycles, recursive terms will not occur.

However, if some functor is in a cycle, this is not conclusive evidence of recursive structure, because the sequence of unifications around the cycle may not actually compose.

**Example 8.2.** Suppose the only nontrivial rules are

\[
p(f(X), g(X)) \leftarrow q(X, X).
\]

\[
q(U, V) \leftarrow p(U, V).
\]

Then the functor precedence graph is

```
   f
   v
```

```
g
```

```
U   X   V
```

Now both \( f \) and \( g \) are in cycles, yet recursive terms will not occur. To see why, make a "ghost" copy of the first rule

\[
p(f(W), g(W)) \leftarrow q(W, W).
\]
The substitution to resolve the first two rules on $q$ is $\theta_1 = \{ U := X; V := X \}$, which is read “replace $U$ by $X$; replace $V$ by $X$”. The substitution to resolve the second rule with the “ghost” on $p$ is $\theta_2 = \{ U := f(W); V := g(W) \}$. Since these substitutions cannot be composed into single substitution, the apparent loop will never be realized.

9. CONCLUSION

We have presented a new semantics, the tight tree semantics, for general logic programs that has advantages over the rule-based semantics in that a larger portion of the Herbrand base tends to be classified as belonging to either the success set or the failure set. We have defined a reasonable class of programs, namely those that have the bounded-term-size property and are free of recursive negation. We have shown that for this class the entire Herbrand base is classified as success or failure, and that an equivalent iterated-fixed-point semantics exists. Further work is needed to extend these results to (certain) programs without the bounded-term-size property.

We thank Teodor Przymusinski and Hirohisa Seki for helpful comments on earlier drafts of this paper.

REFERENCES


