Chapter

2

Towards a Theory of Declarative Knowledge

Krzysztof R. Apt[1]
C.W.I.,
Amsterdam, The Netherlands

Howard A. Blair[2]
CIS,
Syracuse University,
Syracuse, NY

Adrian Walker
IBM Thomas J. Watson Research Center,
Yorktown Heights, NY

Abstract

We identify a useful class of logic programs with negation, called *stratified programs*, that disallow certain combinations of recursion and negation. Programs in this class have a simple declarative and procedural meaning based, respectively, on model theory and a back-chaining interpreter. The standard model of a stratified program, which gives the program a declarative meaning

[1]Work performed during the author's visiting year at IBM Research Center, Yorktown Heights, NY 10598

[2]Work performed while the author was a consultant at IBM Research Center, Yorktown Heights, NY 10598

and is independent of the stratification, is characterized in two ways. One is based on a fixed point theory of nonmonotonic operators and the other on an abstract declarative characterization. The back-chaining interpreter also determines the standard model. Finally, we prove the consistency of Clark's completion for stratified programs and attempt to clarify the sources of some previously reported difficulties with negation in logic programming.

Introduction

The aim of this paper is to provide a formal basis for separating declarative and procedural matters in an extension of logic programming allowing negation in the presence of certain recursions. This should be viewed as part of a larger research program with the aim of extending logic programming so that it is more useful for expert systems, both as a formal basis and as a source of practical techniques.

Expert Systems

There is currently considerable interest in expert system shells that are based on logic programming. Logic programming certainly has several properties that should be seriously considered while searching for an appropriate formalism. Among other things, we expect that the knowledge in an expert system should be easy to examine and to change, and we expect the system to provide explanations of its results. At first sight, the knowledge in a logic program is easy to change, since it consists simply of facts and rules. From the *declarative* point of view, this is indeed the case. However, a common difficulty is that the addition of a rule with an intended declarative meaning has unintended *procedural* effects when the knowledge is interpreted by a particular inference engine. For example, the rule

(X is married to Y) if (Y is married to X)

has an obvious common sense declarative meaning, but it can cause control problems for several well-known inference mechanisms, including that of Prolog.

Similar issues arise in systems that are not based on logic, but logic as a formalism for expert systems has an important advantage: We can specify the *declarative meaning* of a collection of rules and facts using the techniques of model theory. This tells us what the consequences of the knowledge should be, independent of particular mechanisms for interpreting the knowledge. We can also specify various *procedural interpretations*, such as SLD resolution, and we can study the extent to which the interpretations live up to our model theoretic standard.

The extent to which an interpreter (or inference engine) behaves according to the declarative reading of knowledge can be crucial to its use by non-programmers, who wish to add knowledge to an expert system. Real world tasks about which knowledge is to be acquired can be complicated. It helps to manage this complexity if the declarative and procedural concerns can be separated. For example, order does not matter for declarative knowledge, but procedurally we might have to find just one of the possible orderings of a set of rules to make certain inference engines behave the way we wish.

At the higher, declarative level it should be sufficient to write down the knowledge correctly in the form of facts and rules. At the lower level, an inference engine should be available to handle the procedural and computational aspects, and to produce the intended declarative meaning.

Logic Programming

For any choice of formalism to succeed, it must be sufficiently expressive for the purposes at hand. Logic programming with definite clauses (see Kowalski [1978]) does not seem to be expressive enough for expert system shells. In particular, we need to express negation.

It is important to realize that use of negation indeed increases the expressive power of logic programming. This may sound paradoxical since, as is well known (see e.g., Tärnlund [1977]), logic programs without negation have the full power of recursion theory. But the point is that in many situations we compute over a finite domain, and this drastically changes the situation.

The realization of this fact led to extensive studies, started by Clark [1978], of the extensions of logic programming incorporating the use of negation. Unfortunately several difficulties have been revealed (see e.g., Shepherdson [1984, 1985, 1988]), and it seems fair to say that so far no satisfactory theory of negation has been proposed. In particular no positive result concerning the use of negation in the presence of recursion has been proved.

A certain amount is known about interpreters that behave according to model theory. For the case of logic programs without negation, both the declarative reading—through the least Herbrand model, and the procedural reading—through SLD resolution, are available and by the results of Apt and van Emden [1982], they naturally correspond through a completeness result. Brough and Walker [1984] showed that, even for function-free ("database like") programs without negation, a strong form of completeness cannot be achieved by any strictly top-down inference engine. Walker [1986b] described an implemented inference engine that uses a mixed top-down/bottom-up strategy that appears to overcome some of this difficulty, even with negation allowed.

Consider the treatment of negation in Prolog. Prolog is normally augmented with a definition of negation that says that $\neg P$ is true if we cannot prove P. This allows programs containing negation in the premises of rules to be executed. However, such programs must at present be written by programmers with an intimate knowledge of the Prolog interpreter's operation. We lack a declarative reading and have to rely on a highly intricate procedural interpretation instead.

One of the difficulties concerning the use of negation in logic programming is that this is an (almost classic) example of nonmonotonic reasoning. Indeed, suppose that by some means we infer from the logic program P a negative fact, say $\neg A$. Naturally we do not expect that $\neg A$ can then be inferred from P augmented by the fact A—otherwise a contradiction could be derived from P and A. Thus, the provability relation is no longer monotonic, and this makes it difficult to study.

Structure of the Paper

In this paper we achieve our goal by restricting the use of negation. We allow both recursion and negation, but we disallow recursion "through negation" as in

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

and we call the resulting programs *stratified*. They are formally defined in the section "Stratified Programs" below. These programs form a simple generalization of a class of programs introduced in the context of the deductive databases by Chandra and Harel [1985].

The declarative meaning of a stratified program is given in a semantic fashion—by certain of its minimal models. (A model is minimal if it has no proper subset that is also a model.) To see why we need minimality, consider the program $p \leftarrow p$. This has models $\{p\}$ and the empty set. $\{p\}$ is not minimal. We rule it out, since there is no way of proving p using the rule. The minimal models that we consider are those that are *supported*, in the sense that each item in such a model is either a fact in the program, or is the conclusion of a ground instance of a rule whose body is true in the model. To see why it is reasonable to require support, consider the program consisting of just the rule $p \leftarrow \neg q$. This program has minimal models $\{p\}$ and $\{q\}$, but only $\{p\}$ is supported. We rule $\{q\}$ out, on the grounds that there is no way of proving q using the rule as it stands.

To study models of logic programs we relate them in the section "From Models to Fixed Points" to fixed points of a natural operator originally introduced in van Emden and Kowalski [1976]. Unfortunately, in the presence of negation this operator is nonmonotonic and can have no fixed points. To resolve the difficulty we develop in the fifth section a fixed point theory of nonmonotonic operators, and in the section "Model Theory of Stratified Programs" apply it to the study of models of stratified programs. We believe that this theory can have other applications in the area of nonmonotonic reasoning.

The declarative meaning of a stratified program is given by exhibiting a particular supported minimal model that can be defined in a simple way by using the T operator of van Emden and Kowalski [1976]. This provides a logical interpretation of negation that also works procedurally.

The procedural reading of knowledge written in a form of a stratified program is provided by defining a top-down interpreter that makes rather simple use of bottom-up information. We define it in a recursive fashion and show in the two sections on the elementary interpreter and its existence that it admits a well-founded inductive definition when applied to stratified programs.

Then we show that the interpreter computes the chosen model of a stratified program, and that in the absence of function symbols the computation is effective and terminating. Finally, in the section "Other Views of Negation and Stratified Programs," we attempt to clarify the negation problem in logic programming and compare our treatment of negation with two other views proposed in the literature—those of Reiter [1978] and Clark [1978]. There we prove the consistency of Clark's completed database for stratified programs, again employing the fixed point techniques. We also explain why the unrestricted use of function symbols in general makes any interpreter nonterminating. The paper concludes by discussing other related work in the final section.

Preliminaries

In this section we recall the basic definitions concerning logic programs. Our only departure from customary treatment of the subject is that we study these programs in the presence of negation. Nothing will be said here on the computation process—only syntax and semantics will be discussed. We start by defining the syntax.

Syntax

We consider here a *first-order language* whose formulas are denoted by S. Its variables are denoted by x, y, and z terms by s, t, and atomic formulas (usually called *atoms*) in turn by the letters A, B, and C. An atom is called *ground* if no variable occurs in it. A *literal* is an atom A or its negation $\neg A$ and is denoted by the letter L. An *atom* is a *positive literal*, and the negation of an atom is a *negative literal*. A *clause* is a formula of the form

$$A \leftarrow L_1 \& \ldots \& L_m$$

where A is an atom, $L_1,\dots,L_m$ are literals and $m \geq 0$. A is the *head* (conclusion) of the clause and $L_1\&\dots\&L_m$ its *body* (hypothesis). Thus, negation is allowed in the body of a clause but not in its head. If m = 0 then, the clause is simply A and is called a *fact*. Otherwise it is called a *rule*. Finally a *program* is a finite set of clauses. A program whose clauses do not contain negation is called *positive*. In other words, in a positive program only positive literals occur in the bodies of the clauses. Such clauses are usually called *definite clauses*. We define *ground (P)* to be the set of all variable-free instances of clauses in P. Note that *ground (P)* depends on the underlying first-order language.

If a program P contains an atom $r(t_1,\dots,t_n)$, then r is a *relation* of P. We make the convention that no relation symbol occurs with different arities in P.

A *goal* is a formula of the form $\leftarrow L_1\&\dots\&L_m$ where $L_1,\dots,L_m$ are literals and $m \geq 0$.

A *substitution*

$$\theta \equiv (t_1/x_1)\dots(t_m/x_m)$$

is defined as usual: It replaces all free occurrences of the variables $x_1,\dots,x_m$ by the terms $t_1,\dots,t_m$, respectively. The replacement is performed simultaneously. $S\theta$ is the result of applying the substitution θ to the formula S. $S\theta$ is called an *instance* of S.

Semantics

The language of a program P is the first-order language determined by all and only the logical symbols occurring in P. The *Herbrand base* U_L of a first-order language L is defined as the set of all variable-free atoms of L. An *interpretation* for L is a subset of the Herbrand base of L. When L is the language of program P we may refer to U_P, the Herbrand base of P, and to interpretations for P. This definition for Herbrand interpretations will be convenient in later sections. Also we discuss the interaction between operators on interpretations of distinct but closely related programs.

The *truth* of a formula in an interpretation is defined as usual: Only those variable-free instances of atoms that are in I are considered to be true in I. A formula is *closed* if it contains no occurrence of a free variable. Formally, we proceed by induction.

DEFINITION 1

Let I be an interpretation.

1. A formula S is, true in I iff each of its closed instances is true in I, that is, for each x occurring free in S, and each variable-free term t,$S(t/x)$ is true in I.

2. A closed atom A is true in I iff $A \in I$.
3. A closed formula $\neg S$ is true in I iff S is not true in I.
4. A closed formula $\exists x.S$ is true in I iff for some variable free term t the formula $S(t/x)$ is true in I.
5. A closed formula $\forall x.S$ is true in I iff (by (1)!) S is true in I.
6. A closed formula $S_1 \leftarrow S_2$ is true in I iff S_2 is not true in I or S_1 is true in I.
7. A closed formula $S_1 \& \ldots \& S_m$ is true in I iff each of the S_i is true in I.
8. A closed formula $S_1 \vee \ldots \vee S_m$ is true in I iff one of the S_i is true in I.
9. A closed formula $S_1 \leftrightarrow S_2$ is true in I iff (S_1 is true in I if and only if S_2 is true in I). ■

An interpretation M is a *model* for Γ if each formula in Γ is true in M (denoted $M \models \Gamma$). If Γ has a model, then Γ is *consistent*. The models here considered are usually called *Herbrand models*. It should be pointed out that they are not the most general models. Consequently, the notion of consistency we use here is a priori stronger than the usual one since it refers to the existence of a Herbrand model only. It is an important aspect of Herbrand's theorem that the two notions of consistency coincide for clauses.

If a model M of Γ is a subset of every other model of Γ, then we say that M is a *least* model of Γ. If M is a model of Γ such that no model of Γ is its proper subset, then we say that M is a *minimal* model of Γ. Thus, a least model is a minimal model, but not necessarily conversely.

Finally, we say that an interpretation I of a program P is *supported* if for each $A \in I$ there exists a clause $A_1 \leftarrow L_1 \& \ldots \& L_m$ in P and a substitution θ such that $I \models L_1\theta \& \ldots \& L_m\theta$, $A = A_1\theta$, and each $L_i\theta$ is ground. Thus, I is supported iff for each $A \in I$ there exists a clause in ground (P) with head A whose body is true in I.

Stratified Programs

We will now propose a treatment of negation in logic programming, which should be a solution to various difficulties exhibited in the literature. It is a-chieved through restricting its use and by proposing a new semantic interpretation. In the section on the existence of the interpreter we justify this semantic definition by proof theoretic means.

Our view of a safe use of negation is the following. When using negation we should refer to an *already known* relation. More specifically, first some relations should be defined (perhaps recursively) in terms of themselves *without* the use of negation. Next, some new relations can be defined in terms of themselves without the use of negation and in terms of the previous ones, possibly *with* the use of negation. This process can be iterated.

In such a way, from the semantic point of view we only negate relations whose meaning is fixed beforehand. It seems that most of the paradoxes concerning negation in logic programming violate this principle.

More precisely, we introduce the following definitions.

DEFINITION 2

Let P be a program.

1. We say that the relation p *refers to* the relation r if there is a clause in P with p on its left-hand side and r on its right-hand side.
2. By a *definition* of a relation symbol r we mean the subset of P consisting of all clauses with a formula on the left side whose relation symbol is r.
3. A relation symbol r occurs *positively* in a positive literal and *negatively* in a negative literal. ■

These definitions formalize intuitive notions frequently used informally. We now provide the central definition of this section.

DEFINITION 3

A program P is called *stratified* if there is a partition

$$P = P_1 \dot{\cup} \ldots \dot{\cup} P_n$$

such that the following two conditions hold for i = 1,...,n:

1. if a relation symbol occurs positively in a clause in P_i, then its definition is contained within $\bigcup\limits_{j \leq i} P_j$.
2. if a relation symbol occurs negatively in a clause in P_i, then its definition is contained within

$$\bigcup_{j<i} P_j.$$

P_1 can be empty.

We say then that P is *stratified by* $P_1 \dot{\cup} \ldots \dot{\cup} P_n$ and each P_i is called a *stratum* of P. Thus, each stratum defines new relations in terms of itself only positively and in terms of the relations from the previous strata, possibly negatively. We let $\bar{P}_i$ denote $P_1 \dot{\cup} \ldots \dot{\cup} P_i$. Thus $\bar{P}_n = P$. ■

EXAMPLE 1

1. Let P be the following program:

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

$$r,$$

$$q \leftarrow q \& \neg r.$$

Then P is stratified by

$$P = \{r\} \cup \{q \leftarrow q \& \neg r\} \cup \{p(x) \leftarrow \neg q\}.$$

2. Let P be the following program:

$$p \leftarrow q,$$

$$q \leftarrow \neg p.$$

Then P is not stratified. Because of the second clause, the definition of p has to appear in a lower stratum than the definition of q. But p refers to q, so condition (1) cannot be satisfied.

The last example suggests the following simple test whether a program is stratified.

DEFINITION 4

By the *dependency graph* of a program P we mean the directed graph representing the relation *refers to* between the relation symbols of P. Formally, *p refers to q* in P iff there is a clause C in P in which p is the relation symbol in the head of C and q is the relation symbol of a literal in the body of C. Note that it may be that p refers to q via several clauses in P. In particular, for any pair of relation symbols p, q there is at most one edge (p,q) in the dependency graph of P. An edge (p,q) is *positive* [*negative*] iff there is a clause C in P in which p is the relation symbol in the head of C, and q is the relation symbol of a positive [negative] literal in the body of C. Note that an edge may be both positive and negative. ■

LEMMA 1

A program P is stratified iff in its dependency graph there are no cycles containing a negative edge.

Proof: If the program is stratified, then the definition of each relation symbol is contained in some stratum. Assign to each relation the index of the stratum within which it is defined. If (p,q) is a positive edge in the dependency graph of P then the level assigned to q is smaller than or equal to that assigned to p, and if (p,q) is a negative edge, then the level assigned to q is strictly smaller than that assigned to p. Thus, there are no cycles in the dependency graph through a negative edge.

For the converse, decompose the dependency graph into *strongly connected* components, each of maximum cardinality (i.e., such that any two nodes in a component are connected in a cycle). Then the relation "there is an edge from component G to component H" is *well founded*, since it is finite, and contains no cycles. Thus, for some n the numbers 1,...,n can be assigned to the components so that if there is an edge from G to H, then the number assigned to H is smaller than that assigned to G. Now let P_i be the subset of the program P consisting of the definitions of all relations which lie within a component with the number i.

We claim that

$$P = P_1 \dot{\cup} \ldots \dot{\cup} P_n$$

is a stratification of P. Indeed, if q is defined within some P_i and refers to r, then r lies in the same component or in a component with a smaller number. In other words, the definition of r is contained in P_j for some $j \leq i$. And if this reference is negative, then r lies in a component with a smaller number because by assumption there is no cycle through a negative edge. Thus, the definition of r is then contained in P_j for some $j < i$. ■

We will now study semantics of stratified programs. As explained in the introduction we are interested in models that are both minimal and supported. Our task is to prove that stratified programs indeed have such models.

From Models to Fixed Points

Van Emden and Kowalski [1976] proposed an elegant way of studying logic programs without negation. Their definitions still make perfect sense in the presence of negation. Their idea was to use a natural closure operator and equate the models of a program P with the pre-fixed points of the operator, which are simpler to analyze. This operator is usually called T_P. It maps interpretations of P into interpretations of P and is defined as follows:

$$A \in T_P(I) \text{ iff for some clause } A_1 \leftarrow L_1 \,\&\ldots\&L_m \text{ in } P \text{ and substitution } \theta,$$
$$I \models L_1\,\theta \,\&\ldots\&L_m\,\theta \text{ and } A = A_1\theta$$

We shall drop the subscript P if it is clear from the context to which program P the operator refers. Intuitively, $T_P(I)$ is the set of immediate conclusions of I, i.e., those which can be obtained by applying a rule from P only once. Note that $A \in T_P(I)$ iff there exists a clause in ground (P) with head A whose body is true in I.

Below, we shall study T as an operator, so it is perhaps useful to recall some terminology and well-known results. Since no greater generality is needed, we consider only the case of operators on complete lattices. We use $\subseteq$ to denote the order relation on the lattice.

We say that T is *monotonic* if $I_1 \subseteq I_2$ implies $T(I_1) \subseteq T(I_2)$. When $T(I) \subseteq I$ then we say that I is a *pre-fixed point* of T and when $T(I) = I$ then we say that I is a *fixed point* of T. The following classical result is at the heart of the van Emden and Kowalski [1976] approach.

THEOREM (KNASTER-TARSKI [1955])
A monotonic operator T has a least fixed point that is also the least pre-fixed point of T. ■

Now suppose that T is one of the operators T_P on Herbrand interpretations. The Knaster-Tarski theorem's importance derives from the fact that for a positive program P the operator T_P is monotonic. Unfortunately, in the presence of negation, things change. In fact, in this paper we shall not make use of the Knaster-Tarski theorem. The following example explains the difficulties.

EXAMPLE 2
Let P be a program without negation. Then (see van Emden and Kowalski [1976]),

a. T_P is monotonic,

b. the intersection of two models of P is a model of P, and

c. P has a least model.

On the other hand, if P is a program with negation we have

d. T_P does not need to be monotonic,

e. Intersection of two models of P does not need to be a model of P, and

f. P may have no least model.

To see this, consider the program P: $A \leftarrow \neg B$

1. Take I_1 empty and $I_2 = \{B\}$. Then $T_P(I_1) = \{A\}$ whereas $T_P(I_2)$ is empty. Thus $I_1 \subseteq I_2$ but not $T_P(I_1) \subseteq T_P(I_2)$.

2. Take the above program P. Then $\{A\}$ and $\{B\}$ are models of P but their intersection is not.

3. P has two different minimal models: $\{A\}$ and $\{B\}$. ■

Several results concerning positive programs depend critically on the properties (a), (b), and (c). Fortunately, a very important property remains true.

LEMMA 2
Let P be a program. Then I is a model of P iff $T_P(I) \subseteq I$.

Proof: Essentially the same as the proof for programs without negation; see Lloyd [1984]. ■

This simple fact saves the whole approach based on the analysis of the operator T_P! Further, the notion of a supported model can also be naturally expressed in terms of the operator T_P. We have the following simple result.

LEMMA 3
Let P be a program. Then I is supported iff $T_P(I) \supseteq I$.

Proof: Direct from the definition. ■

As explained in the introduction, we are interested here in studying minimal and supported models. In view of Lemmas 2 and 3 this simply means that we are looking for minimal fixed points of the operator T_P. Thus, we are brought to study the fixed points of nonmonotonic operators. It is conceptually advantageous to carry out such an analysis in a completely general situation.

Fixed Point Theory of Nonmonotonic Operators

Powers

Here we study operators over an arbitrary, but fixed, complete lattice. To keep in mind the subsequent applications to logic programs and their interpretations, we denote the least element by ϕ and the elements of the lattice by I, J, M. The order relation on the lattice is denoted by $\subseteq$.

We start by defining *powers* of an operator T. We put

$$T\uparrow 0(I) = I$$

$$T\uparrow(n+1)\,(I) = T(T\uparrow n(I)) \cup T\uparrow n(I).$$

$$T\uparrow\omega(I) = \bigcup_{n=0}^{\infty} T\uparrow n(I)$$

$T\uparrow n(I)$ should not be confused with $T^n(I)$, which stands for the n fold application of T. If T is monotonic, then its ordinal powers can be (and usually are) defined in a slightly simplified way by putting

$$T\uparrow(n+1)\,(I) = T(T\uparrow n(I))$$

An obvious proof by induction shows that when T is monotonic both definitions lead to the same value of $T\uparrow n(\phi)$ (but of $T\uparrow n(I)$ only when $I \subseteq T(I)$).

Clearly the process of computing powers of T can be extended beyond ω, see Blair [1982], but we shall not need this.

Fixed Points

We now introduce the following definition.

DEFINITION 5
T is *finitary* if for every infinite sequence

$$I_0 \subseteq I_1 \subseteq \cdots$$
$$T\left(\bigcup_{n=0}^{\infty} I_n\right) \subseteq \bigcup_{n=0}^{\infty} T(I_n)$$

holds. Thus if $A \in T(\bigcup_{n=0}^{\infty} I_n)$ then for some n, $A \in T(I_n)$, which explains the name. The following lemma shows the importance of this notion. ∎

LEMMA 4
If T is finitary then for all I

$$T(T\uparrow\omega(I)) \subseteq T\uparrow\omega(I).$$

Proof: We have

$$T(T\uparrow\omega(I))$$
$$\subseteq \bigcup_{n=0}^{\infty} T(T\uparrow n(I)) \text{ since } T \text{ is finitary}$$
$$\subseteq T\uparrow\omega(I) \text{ since } T(T\uparrow n(I)) \subseteq T\uparrow(n+1)(I). \ \blacksquare$$

Thus, finitary operators have pre-fixed points which can be computed in a natural way. In general finitary operators do not have fixed points, but under some assumptions they do.

DEFINITION 6
T is *growing* if for all I, J, M

$$I \subseteq J \subseteq M \subseteq T\uparrow\omega(I)$$

implies

$$T(J) \subseteq T(M). \blacksquare$$

The following lemma holds.

LEMMA 5
If T is growing then for all I

$$T\uparrow\omega(I) \subseteq I \cup T(T\uparrow\omega(I))$$

Proof: We have

$$\begin{aligned} A \in T\uparrow\omega(I) &\Rightarrow A \in I \text{ or for some } n \geq 1, A \in T\uparrow n(I) \\ &\Rightarrow A \in I \text{ or for some } n \geq 0, A \in T(T\uparrow n(I)) \\ \text{(by assumption) } &\Rightarrow A \in I \text{ or } A \in T(T\uparrow\omega(I)). \blacksquare \end{aligned}$$

COROLLARY 1
If T is finitary and growing then

$$T\uparrow\omega(\phi) = T(T\uparrow\omega(\phi)). \blacksquare$$

Thus for finitary and growing T, $T\uparrow\omega(\phi)$ is a fixed point.

Iterations

Next, we study finite families of operators. Let $T_1,\ldots,T_n$ be operators. We put

$$N_0 = I,$$
$$N_1 = T_1\uparrow\omega(N_0),$$
$$\ldots$$
$$N_n = T_n\uparrow\omega(N_{n-1}).$$

Clearly $N_0 \subseteq N_1 \subseteq \ldots \subseteq N_n$. Of course all N_i depend on I and it will be always clear from the context from which one. To concentrate attention on the fact that N_n is computed using T_i in an iterative fashion, we sometimes denote it by $iter(T_1,\ldots,T_n,(I)$.

Our first task is to determine under which conditions $iter(T_1,\ldots,T_n,(I)$ is a fixed point of $\bigcup_{i=1}^{n} T_i$, where $(\bigcup_{i=1}^{n} T_i)(X) = \bigcup_{i=1}^{n} (T_i(X))$. For this purpose we introduce the following concept.

DEFINITION 7
A sequence of operators $T_1,\ldots,T_n$ is *local* if for all I,J and $i = 1,\ldots,n$

$$I \subseteq J \subseteq N_n$$

implies

$$T_i(J) = T_i(J \cap N_i).$$

Informally, locality means that each T_i is determined by its values on the subsets of N_i.

As an example of a non-local sequence of operators, consider T_{P_1}, T_{P_2} where $P_1 = \{q \leftarrow \neg p\}$ and $P_2 = \{p \leftarrow \neg p\}$. Then $I = \phi$, $N_1 = \{q\}$, and $N_2 = \{p,q\}$. Choose $J = \{p\}$. Then $T_{P_1}(J) = \phi$ but $T_{P_1}(J \cap N_1) = \{q\}$. ■

We have the following two lemmas.

LEMMA 6
Suppose that the sequence $T_1,\ldots,T_n$ is local and that all T_i are finitary. Then

$$(\bigcup_{i=1}^{n} T_i)(iter(T_1,\ldots,T_n, I)) \subseteq iter(T_1,\ldots,T_n,I).$$

Proof: We have

$$\begin{aligned} &\bigcup_{i=1}^{n} T_i(iter(T_1,\ldots,T_n, I)) \\ \text{(by locality)} = &\bigcup_{i=1}^{n} T_i(N_i) \\ \text{(by Lemma 4)} \subseteq &\bigcup_{i=1}^{n} N_i \\ = &\; iter(T_1,\ldots,T_n, I). \end{aligned}$$ ■

LEMMA 7
Suppose that the sequence $T_1,\ldots,T_n$ is local and each T_i is growing. Then

$$iter(T_1,\ldots,T_n,I) \subseteq I \cup (\bigcup_{i=1}^{n} T_i)(iter(T_1,\ldots,T_n,I)).$$

Proof: We prove it by induction on n. If $n = 1$, the lemma reduces to Lemma 5. For $n > 1$, assume the lemma holds for all $m < n$. Then

$$\begin{array}{ll} & N_n \\ \text{(by Lemma 5)} & \subseteq N_{n-1} \cup T_n(N_n) \\ \text{(by ind. hypothesis)} & \subseteq (I \cup (\bigcup_{i=1}^{n-1} T_i)(N_{n-1})) \cup T_n(N_n) \\ \text{(by locality)} & = I \cup (\bigcup_{i=1}^{n} T_i)(N_n). \ \blacksquare \end{array}$$

COROLLARY 2

Suppose that sequence $T_1,\ldots,T_n$ is local and all T_i are finitary and growing. Then

$$iter(T_1,\ldots,T_n, I) = I \cup (\bigcup_{i=1}^{n} T_i)(iter(T_1,\ldots,T_n, I)). \ \blacksquare$$

Thus for a local sequence $T_1,\ldots,T_n$ of finitary and growing operators $iter(T_1,\ldots,T_n, \phi)$ is a fixed point of $\bigcup_{i=1}^{n} T_i$.

We now prove that under some assumptions $iter(T_1,\ldots,T_n, I)$ is a minimal pre-fixed point of $\bigcup_{i=1}^{n} T_i$ containing I. More precisely, we prove

THEOREM 1

Suppose that the sequence $T_1,\ldots,T_n$ is local and that all T_i are growing. If

$$I \subseteq J \subseteq iter(T_1,\ldots,T_n, I)$$

and

$$(\bigcup_{i=1}^{n} T_i)(J) \subseteq J$$

then

$$J = iter(T_1,\ldots,T_n, I).$$

Proof: We prove by induction on j = 0,...,n that

$$N_j \subseteq J. \tag{1}$$

For j = 0 it is part of the assumptions. Assume that the claim holds for some $j < n$. We now prove by induction on k that

$$T_{j+1} \uparrow k(N_j) \subseteq J. \tag{2}$$

For k = 0 this is just (1), So assume (2) holds for some $k \geq 0$. We have

$$\begin{aligned} & T_{j+1}\uparrow(k+1)(N_j) \subseteq T_{j+1}\,(T_{j+1}\uparrow k(N_j)) \cup J \\ \text{(by (2) and since } T_{j+1} \text{ is growing)} \quad & \subseteq T_{j+1}(J \cap N_{j+1}) \cup J \\ \text{(by locality)} \quad & = T_{j+1}(J) \cup J \\ \text{(by the assumptions)} \quad & \subseteq J. \end{aligned}$$

Thus by induction for all k (2) holds, so $N_{j+1} \subseteq J$. This proves (1) for all j = 0,...,n and concludes the proof. ■

Iteration Versus Simultaneity

Next, we relate $iter(T_1,\ldots,T_n,I)$ with $(\bigcup_{i=1}^{n} T_i)\uparrow\omega(I)$. We need the following notion.

DEFINITION 8
A sequence of operators $T_1,\ldots,T_n$ is *raising* if for all I,J,M and $i = 1,\ldots,n$

$$I \subseteq J \subseteq M \subseteq N_n$$

implies

$$T_i(J) \subseteq T_i(M). \ ■$$

We have the following lemma.

LEMMA 8
Suppose that the sequence $T_1,\ldots,T_n$ is local and raising and that all T_i are finitary. Then

$$(\bigcup_{i=1}^{n} T_i)\uparrow\omega(I) \subseteq iter(T_1,\ldots,T_n,I).$$

Proof: We prove by induction on k that

$$(\bigcup_{i=1}^{n} T_i)\uparrow k(I) \subseteq iter(T_1,\ldots,T_n,I) \qquad (3)$$

It clearly holds for $k = 0$. So assume (3) holds for some $k \geq 0$. To make the derivation more readable, denote $(\bigcup_{i=1}^{n} T_1)\uparrow k(I)$ by J. We have

$$(\bigcup_{i=1}^{n} T_i \uparrow (k+1)(I) = (\bigcup_{i=1}^{n} T_i)(J) \cup J$$

(by locality and (3) $\quad = (\bigcup_{i=1}^{n} T_i)(J \cap N_i) \cup J$

(since $T_1,\ldots,T_n$ is raising) $\subseteq (\bigcup_{i=1}^{n} T_i)(N_i) \cup J$

(by Lemma 4) $\quad \subseteq \bigcup_{i=1}^{n} N_i \cup J$

(by(3)) $\quad \subseteq iter(T_1,\ldots,T_n, I)$.

Thus, by induction (3) holds for all k, which completes the proof. ■

This leads us to the following theorem.

THEOREM 2

Suppose that the sequence $T_1,\ldots,T_n$ is local and raising. Suppose also that all T_i are finitary and growing. Then

$$iter(T_1,\ldots,T_n, I) = (\bigcup_{i=1}^{n} T_i)\uparrow\omega(I).$$

Proof: This time it suffices to combine previously proved results. First, observe that since all T_i are finitary, $\bigcup_{i=1}^{n} T_i$ is finitary, as well. Thus by Lemma 4

$$(\bigcup_{i=1}^{n} T_i)((\bigcup_{i=1}^{n} T_i)\uparrow\omega(I)) \subseteq (\bigcup_{i=1}^{n} T_i)\uparrow\omega(I).$$

Now, in view of Lemma 8 all assumptions of Theorem 1 are satisfied and the conclusion follows. ■

Independence

Finally, we study a condition under which the order of application of two operators is irrelevant. For this purpose we introduce the following notion.

DEFINITION 9

T_1 and T_2 are *independent* if for all I, J and M

1. if $I \subseteq J \subseteq T_2\uparrow\omega(I)$ then $T_1(I \cup M) = T_1(J \cup M)$,
2. if $I \subseteq J \subseteq T_1\uparrow\omega(I)$ then $T_2(I \cup M) = T_2(J \cup M)$. ■

Intuitively, in (1), T_1 makes no use of $J - I$. Consider $I \subseteq J \subseteq T_2\uparrow\omega(I)$ and $M = \phi$. Then $T_1(I) = T_1(J)$.

We have the following technical lemma.

LEMMA 9
Suppose that T_1 and T_2 are independent. Then for all $k \geq 1$

1. if $I \subseteq J \subseteq T_2 \uparrow \omega(I)$ then $T_1 \uparrow k(J) = T_1 \uparrow k(I) \cup J$,
2. if $I \subseteq J \subseteq T_1 \uparrow \omega(I)$ then $T_2 \uparrow k(J) = T_2 \uparrow k(I) \cup J$.

Proof: Suppose that $I \subseteq J \subseteq T_2 \uparrow \omega(I)$. Note that

$$T_1 \uparrow 1(J) = T_1 \uparrow 1(I) \cup J.$$

Suppose now the claim holds for some $k \geq 1$. We then have (1) by an obvious induction. By symmetry (2) holds as well, which proves the lemma. ■

The lemma implies the following theorem.

THEOREM 3
Suppose that T_1 and T_2 are independent. Then for all I

$$T_1 \uparrow \omega(T_2 \uparrow \omega(I)) = T_2 \uparrow \omega(T_1 \uparrow \omega(I)).$$

Proof: We have

$$\begin{aligned} T_1 \uparrow \omega(T_2 \uparrow \omega(I)) &= \bigcup_{k=1}^{\infty} T_1 \uparrow k(T_2 \uparrow \omega(I)) \\ \text{(by Lemma 9)} \qquad &= \bigcup_{k=1}^{\infty} (T_1 \uparrow k(I) \cup T_2 \uparrow \omega(I)) \\ &= T_1 \uparrow \omega(I) \cup T_2 \uparrow \omega(I) \\ \text{(by symmetry)} \qquad &= T_2 \uparrow \omega(T_1 \uparrow \omega(I)). \ ■ \end{aligned}$$

It is time to relate the above results to logic programs.

Model Theory of Stratified Programs

Consider a program P stratified by

$$P = P_1 \dot{\cup} \ldots \dot{\cup} P_n.$$

We now define a standard interpretation of P by putting

$$M_1 = T_{P_1} \uparrow \omega(\phi),$$
$$M_2 = T_{P_2} \uparrow \omega(M_1),$$
$$\ldots$$
$$M_n = T_{P_n} \uparrow \omega(M_{n-1}).$$

Let $M_P = M_n$.

In what sense is M_P standard? We prove in this and the next sections several results which support the claim that M_P is natural. Obviously our first task is to prove that M_P is a model of P.

This turns out to be an easy consequence of the results proved in the previous section.

Minimality and Supportedness of M_P

We first prove certain facts about the operators associated with logic programs.

THEOREM 4
For all programs P, T_P is finitary.

Proof: Straightforward and left to the reader. ■

Next let us introduce the following useful notation.

$Neg_P = \{A : \neg A$ is a variable-free instance of a negative literal in a clause in $P\}$

$Def_P = \{A : A$ is a variable-free instance of a head of a clause in $P\}$

EXAMPLE 3
Let P be the following program:

$p(a)$,
$r(x) \leftarrow \neg q(a)$,
$p(x) \leftarrow \neg r(y)$.

Then

$$Neg_P = \{q(a), r(a)\}$$

and

$$Def_P = \{p(a), r(a)\}. \ ■$$

We have the following simple lemma.

LEMMA 10

Let P be a subprogram of P'. Then

$$I \subseteq J \subseteq U_{P'} \text{ and } I \cap Neg_P = J \cap Neg_P$$

implies

$$T_P(I) \subseteq T_P(J).$$

Informally, the lemma says that each T_P is monotonic as long as its arguments do not differ on the elements of Neg_P. The reference to P' and $U_{P'}$ allows us to consider T_P on a larger space.

Proof: Suppose that $A \in T_P(I)$. Then there is a variable-free instance of a clause from P of the form

$$A \leftarrow L_1 \& \ldots \& L_m$$

where

$$I \models L_1, \ldots, I \models L_m.$$

If L_i is positive, then $L_i \in I$, so also $L_i \in J$ and consequently $J \models L_i$. If L_i is negative, then it is of the form $\neg B_i$ where $B_i \notin I$. Also $B_i \in Neg_P$. Thus $B_i \notin I \cap Neg_P$ and by the assumption $B_i \notin J \cap Neg_P$. Hence $B_i \notin J$ and consequently $J \models L_i$. Thus for all i, $J \models L_i$ which implies $A \in T_P(J)$. ■

Next we introduce the following definition.

DEFINITION 10

A program is called *semi-positive* if none of its negated relation symbols occurs in a head of a clause. More formally, P is semi-positive if $Neg_P \cap Def_P = \phi$. ■

The following is a consequence of the previous lemma.

THEOREM 5

If P is semi-positive, then T_P is growing.

Proof: First observe that if

$$I \subseteq J \subseteq M \subseteq T_P \uparrow \omega(I)$$

then, since $T_P \uparrow \omega(I) \subseteq I \cup Def_P$, we have

$$M \cap Neg_P \subseteq (I \cup Def_P) \cap Neg_P = I \cap Neg_P \subseteq J \cap Neg_P.$$

so

$$J \cap Neg_P = M \cap Neg_P.$$

Therefore by Lemma 10

$$T_P(J) \subseteq T_P(M). \blacksquare$$

Informally, for semi-positive programs all arguments of T_P lying between I and $T_P \uparrow \omega(I)$ do not differ on the elements from Neg_P.

Now, consider a sequence of programs $P_1,\dots,P_n$.

DEFINITION 11

A sequence $P_1,\dots,P_n$ *defines new relations* if the following holds: whenever a relation symbol occurs in a clause P_i, then its definition within $P_1 \cup \dots \cup P_n$ is contained in P_j for some $j \leq i$. (Note that if the definition of a given relation symbol is empty, then that definition is contained within, in particular, P_1.)

More formally, a sequence $P_1,\dots,P_n$ defines new relations if for all i = 1 ,…,n − 1

$$Def_{P_{i+1}} \cap U_{P_1 \cup \dots \cup P_i} = \phi. \blacksquare$$

That is, none of the relation symbols defined in P_{i+1} is mentioned in $P_1,\dots,P_i$.

We have the following theorem.

THEOREM 6

If the sequence $P_1,\dots,P_n$ defines new relations, then the sequence of the operators $T_{P_1},\dots,T_{P_n}$ considered on the space $U_{P_1 \cup \dots \cup P_n}$ is local.

Proof: As in the previous section we denote $iter(T_{P_1},\dots,T_{P_i}, I)$ by N_i and let N_0 be I. We have for $i = 0,\dots n-1$

$$N_{i+1} - N_i \subseteq Def_{P_{i+1}}.$$

Thus for $i = 1,\dots,n$

$$\begin{aligned} & N_n \cap U_{P_1 \cup \dots \cup P_i} \\ & \subseteq (N_i \cup Def_{P_{i+1}} \cup \dots \cup Def_{P_n}) \cap U_{P_1 \cup \dots \cup P_i} \\ \text{(since } P_1,\dots,P_n \text{ defines new relations)} \quad & \subseteq N_i \cap U_{P_1 \cup \dots \cup P_i} \\ & \subseteq N_n \cap U_{P_1 \cup \dots \cup P_i}. \end{aligned}$$

Hence

$$N_i \cap U_{P_1 \cup \ldots \cup P_i} = N_n \cap U_{P_1 \cup \ldots \cup P_i}$$

Suppose now that

$$I \subseteq J \subseteq N_n.$$

Then for i = 1,...,n

$$\begin{aligned} J \cap U_{P_1 \cup \ldots \cup P_i} &= J \cap N_n \cap U_{P_1 \cup \ldots \cup P_i} \\ &= J \cap N_i \cap U_{P_1 \cup \ldots \cup P_i}. \end{aligned}$$

But by the definition of T_P

$$\begin{aligned} T_{P_i}(J) &= T_{P_i}(J \cap U_{P_1 \cup \ldots \cup P_i}) \\ &= T_{P_i}(J \cap N_i \cap U_{P_1 \cup \ldots \cup P_i}) \\ &= T_{P_i}(J \cap N_i) \end{aligned}$$

as desired. ■

Let us now relate the above theorems to stratfied programs. Suppose that P is stratified by

$$P = P_1 \dot{\cup} \ldots \dot{\cup} P_n.$$

Then by definition each P_i is semi-positive and the sequence $P_1,\ldots,P_n$ defines new relations. By Theorems 4 and 5 all operators T_{P_i} for $i = 1,\ldots n$ are finitary and growing. By Theorem 6 the sequence $T_{P_1},\ldots T_{P_n}$ is local. We are now in a position to apply Corollary 2. It implies

$$T_P(M_P) = M_P.$$

This in turn, in view of Lemmas 2 and 3, implies the following theorem.

THEOREM 7

1. M_P is a model of P.
2. M_P is supported. ■

Also, by Theorem 1 and Lemma 2 we have

THEOREM 8
M_P is a minimal model of P. ■

Note that in view of Lemmas 6 and 2 and Theorems 4 and 6, M_P is a model of P whenever the sequence $P_1,\ldots,P_n$ defines new relations. But to prove that M_P is supported we also need Lemma 7, which requires that all T_{P_i} are growing. This condition is satisfied (see Theorem 5) if each P_i is semi-positive. Now it is easy to see that if the sequence $P_1,\ldots,P_n$ defines new relations and each P_i is semi-positive, then P is stratified by $P = P_1 \dot{\cup} \ldots \dot{\cup} P_n$. In other words, our general results on nonmonotonic operators do not allow us to conclude existence of supported models for other than stratified programs.

Independence of M_P from the Stratification

The definition of the model M_P for a stratified program P is somewhat unsatisfactory as it explicitly refers to the way P is stratified. We now prove that M_P does not depend on the stratification of P. This will support our claim that M_P is a natural model for P.

Again this easily follows from the results of the previous section. This time we use results of the second part of the section. We first introduce the following natural concept.

DEFINITION 12
Let P be a program.

1. *depends on* is the reflexive transitive closure of the relation *refers to* between the relation symbols of P.
2. By a *cluster* we mean a non-empty subset of P that is the union of a maximal collection of definitions that define relations that depend on one another. ■

The following example should clarify the definition.

EXAMPLE 4
Let P be

$p \leftarrow q,$

$r \leftarrow q \& q_1,$

$q \leftarrow r,$

$q_1.$

Then q and r depend on each other. Thus $\{p \leftarrow q\}$, $\{r \leftarrow q\&q_1, \quad q \leftarrow r\}$ and $\{q_1\}$ are the clusters of P. ■

Note that P' is a cluster if it is non-empty and for each clause in P' with p on its left hand side and r on its right hand side. P' contains the definition of p and, if r depends on p, the definition of r.

We have the following simple lemma.

LEMMA 11

Let P be a program.

1. The clusters of P form a partition of P.
2. If P is stratified by $P_1 \dot{\cup} \ldots \dot{\cup} P_n$, then each stratum P_i is a union of clusters.

Proof: Consider the following relation between definitions from P:

$R_1 \geq R_2$ iff the relation defined by R_1 depends on the relation defined by R_2.

Let

$R_1 \approx R_2$ iff $R_1 \geq R_2$ and $R_2 \geq R_1$.

Then $\approx$ is an equivalence relation between the definitions from P.

To prove (1) it is now sufficient to observe that a cluster is just a union of definitions forming an equivalence class of $\approx$.

To prove (2) note that each definition from P is contained in a stratum. Let *index*(R) for a definition R be the index of the stratum it is contained in. (We shall also say that the index of a literal is the index of the definition of its relation symbol.) Then by the definition of stratification

$R_1 \geq R_1 \Rightarrow \text{index}(R_1) \geq \text{index}(R_2)$.

Thus

$R_1 \approx R_2 \Rightarrow \text{index}(R_1) = \text{index}(R_2)$.

so in view of the above characterization of clusters, each cluster is contained in a stratum. The claim now follows by (1). ■

Consider now the relation $\geq$ defined in the last proof but now as a relation between clusters. In other words, given two clusters Q_1 and Q_2 we put

$$Q_1 \geq Q_2 \text{ iff for some definitions } R_1 \subseteq Q_1$$
$$\text{and } R_2 \subseteq Q_2 \text{ we have } R_1 \geq R_2.$$

Then, since $\approx$ is an equivalence relation, $\geq$ is a partial ordering between clusters.

Given now a program stratified by $P = P_1 \dot{\cup} \ldots \dot{\cup} P_n$, let *index(Q)*, for a cluster Q, be the index of a stratum it is contained in. Then for two clusters Q_1 and Q_2

$$Q_1 \geq Q_2 \Rightarrow \text{index}(Q_1) \geq \text{index}(Q_2).$$

We now introduce the following definition.

DEFINITION 13
We say that two clusters Q_1, Q_2 are *unrelated* if neither $Q_1 \geq Q_2$ nor $Q_2 \geq Q_1$ holds.

Note that if Q_1 and Q_2 are unrelated, then $Def(Q_2) \cap U_{Q_1} = \phi$ and $Def(Q_1) \cap U_{Q_2} = \phi$. but not necessarily conversely. For example, if $P = \{p \leftarrow q, q \leftarrow r, r\}$ then the clusters $Q_1 = \{p \leftarrow q\}$ and $Q_2 = \{r\}$ are not unrelated but satisfy the above property. ■

The following is an immediate consequence of the above remarks.

LEMMA 12
Let P be a stratified program. Suppose that $P = P_1 \dot{\cup} \ldots \dot{\cup} P_k$ and $P = P'_1 \dot{\cup} \ldots \dot{\cup} P'_l$ are two different stratifications of P. If Q_1, Q_2 are two clusters such that for some i_1, i_2, j_1, j_2

$$Q_1 \subseteq P_{i_1}, Q_2 \subseteq P_{i_2}, i_1 < i_2$$

and

$$Q_1 \subseteq P'_{j_1}, Q_2 \subseteq P'_{j_2}, j_1 > j_2$$

then Q_1 and Q_2 are unrelated.

Proof: The conditions of the lemma simply state that the order of indexes of Q_1 and Q_2 in two stratifications of P is different. Thus neither $Q_1 \geq Q_2$ nor $Q_2 \geq Q_1$ holds. ■

We now link the notions of this and the previous sections.

THEOREM 9

Let Q_1, Q_2 be two clusters of a program P. Suppose that Q_1 and Q_2 are unrelated. Then T_{Q_1} and T_{Q_2} are independent.

Proof: Suppose that for some I, J

$$I \subseteq J \subseteq T_{Q_2}\uparrow\omega(I).$$

Then

$$J - I \subseteq Def_{Q_2},$$

so since Q_1 and Q_2 are unrelated.

$$(J - I) \cap U_{Q_1} = \phi.$$

Thus for any M

$$(I \cup M) \cap U_{Q_1} = (J \cup M) \cap U_{Q_1},$$

that is

$$T_{Q_1}(I \cup M) = T_{Q_1}(J \cup M).$$

By symmetry the other half of the independence definition holds as well. ■

THEOREM 10

Let P be a program stratified by $P = P_1 \dot{\cup} \ldots \dot{\cup} P_k$ and suppose that $Q_1,\ldots,Q_n$ are the clusters of P_i for some i, $1 \leq i \leq n$. Then the sequence of operators $T_{Q_1},\ldots,T_{Q_n}$ considered on the space U_{P_i} is raising.

Proof: Since P_i is a stratum, it is a semi-positive program.
Suppose that

$$I \subseteq J \subseteq M \subseteq iter(T_{Q_1},\ldots,T_{Q_n}, I)$$

Then for any $j = 1,\ldots,n$

$$J \cap Neg_{Q_j} \subseteq (Def_{P_i} \cup I) \cap Neg_{Q_j}$$

(since P_i is semi-positive and $Neg_{Q_j} \subseteq Neg_{P_i}$) $= I \cap Neg_{Q_j}$.

Similarly

$$M \cap Neg_{Q_j} \subseteq I \cap Neg_{Q_j}.$$

It follows that

$$I \cap Neg_{Q_j} = J \cap Neg_{Q_j} = M \cap Neg_{Q_j}.$$

Now, by Lemma 10

$$T_{Q_j}(J) \subseteq T_{Q_j}(M). \quad ■$$

Finally we prove the following theorem.

THEOREM 11
Let P be a stratified program. Then M_P is independent of the stratification of P.

Proof: We use the previous two theorems.

Given two stratifications $P_1,\ldots,P_n$ and $P'_1,\ldots,P'_k$ of P, we say that they are *equivalent* if they yield the same model M_P, that is if

$$iter(T_{P_1},\ldots,T_{P_n}, \phi) = iter(T_{P'_1},\ldots,T_{P'_k}, \phi).$$

We now prove that any two stratifications of P are equivalent. So let $P_1,\ldots,P_n$ be a stratification of P. By Lemma 11 (2) each stratum is a union of clusters. Consider a stratum, say P_i, and a sequence of its clusters, say $Q_1,\ldots,Q_h$. This sequence can be rearranged so that for every j,m, $1 \leq j,m \leq h$

$$Q_j \geq Q_m \Rightarrow j \geq m.$$

Now $P_1,\ldots,P_{i-1}$, $Q_1,\ldots,Q_h$, $P_{i+1},\ldots,P_n$ is also a stratification of P. Moreover, by Theorems 6, 10, and 2 applied to T_{Q_1}, $\ldots,T_{Q_h}$ and T_{P_i}, it is a stratification equivalent to the previous one. Iterating this procedure for other strata we arrive at a stratification of P consisting of clusters which is equivalent to the original one. ■

It now suffices to prove that any two stratifications of P consisting of clusters are equivalent. To this purpose we need the following simple lemma.

LEMMA 13
Let $a_1,\ldots,a_n$ and $b_1,\ldots,b_n$ be two permutations of n elements. Then $a_1,\ldots,a_n$ can be transformed into $b_1,\ldots,b_n$ by repeatedly exchanging two adjacent elements whose relative order in $b_1,\ldots,b_n$ is different.

Proof: Straightforward—use the bubble sort. ■

We now complete the proof of Theorem 11. Let $Q_1,\ldots,Q_n$ and $Q'_1,\ldots,Q'_n$ be two stratifications of P consisting of clusters. If the relative order of two clusters Q_i and Q_j in these two sequences differs, then by Lemma 12 Q_i and Q_j are unrelated. Then by Theorem 4, T_{Q_i} and T_{Q_j} are independent. This together with the previous lemma concludes the proof. ■

An Alternative Characterization of M_P

The definition of the model M_P is somewhat operational in the sense that it is defined in terms of the iterations of the operators T_{P_i}. We now offer another, equivalent definition, which while being less direct has the virtue of not referring to any computation mechanism.

Suppose that P is stratified by

$$P = P_1 \dot{\cup} \ldots \dot{\cup} P_n.$$

Recall that $\bar{P}_i$ denotes $P_1 \dot{\cup} \ldots \dot{\cup} P_i$. Put

$$M(\bar{P}_1) = \bigcap\{M : M \text{ is a supported model of } \bar{P}_1\}$$

$$M(\bar{P}_2) = \bigcap\{M : M \cap U_{\bar{P}_1} = M(\bar{P}_1) \text{ and M is a supported model of } \bar{P}_2\}$$

$$\ldots$$

$$M(\bar{P}_n) = \bigcap\{M : M \cap U_{\bar{P}_{n-1}} = M(\bar{P}_{n-1}) \text{ and M is a supported model of } \bar{P}_n\}$$

We now prove the following theorem.

THEOREM 12
$M_P = M(\bar{P}_n)$.

Proof: As expected we proceed by induction and prove that for all $i = 1,\ldots,n$ $M_i = M(\bar{P}_i)$.

For $i = 1$ it is the consequence of the fact that for a monotonic operator T, $T\uparrow k(\phi) = T^k(\phi)$ for all k, and a result of van Emden and Kowalski [1976] characterizing, for a positive program P,

$\bigcup_{k=1}^{\infty} T_P^k(\emptyset)$ as $\bigcap \{M : T_P(M) = M\}$.

Suppose now that the claim holds for some $i < n$. We then have by Theorem 7 applied to $\bar{P}_{i+1}$ that M_{i+1} is a supported model of $\bar{P}_{i+1}$ and, since the sequence $P_1, \ldots, P_{i+1}$ defines new relations,

$$M_{i+1} \cap U_{\bar{P}_i} = M_i \tag{1}$$

(by the induction hypothesis) $= M(\bar{P}_i)$.

Thus, from the definition of $M(\bar{P}_{i+1})$,

$$M(\bar{P}_{i+1}) \subseteq M_{i+1}.$$

To prove that equality actually holds, it is enough to show that $M(\bar{P}_{i+1})$ is a model of $\bar{P}_{i+1}$ and apply Theorem 8, which states that M_{i+1} is a minimal model of $\bar{P}_{i+1}$.

For this purpose we prove

$$T_{\bar{P}_{i+1}}(M(\bar{P}_{i+1})) \subseteq M(\bar{P}_{i+1}) \tag{2}$$

and use Lemma 2.

First note that by (1) and the definition of $M(\bar{P}_{i+1})$ we have

$$M(\bar{P}_{i+1}) \cap U_{\bar{P}_i} = M(\bar{P}_i). \tag{3}$$

Now take M such that $M \models \bar{P}_{i+1}$, M is a supported model of $\bar{P}_{i+1}$ and

$$M \cap U_{\bar{P}_i} = M(\bar{P}_i). \tag{4}$$

Then, by definition, $M(\bar{P}_{i+1}) \subseteq M$ and moreover by (4) $M \cap Neg_{\bar{P}_{i+1}} \subseteq U_{\bar{P}_i}$ as M is a supported model of $\bar{P}_{i+1}$. Thus

$$\begin{aligned} M(\bar{P}_{i+1}) \cap Neg_{\bar{P}_i+1} &= M \cap U_{\bar{P}_i} \cap Neg_{\bar{P}_{i+1}} \\ &= M \cap Neg_{\bar{P}_{i+1}}. \end{aligned}$$

Now by Lemma 10

$$\begin{array}{lrl}
 & T_{\bar{P}_{i+1}}(M(\bar{P}_{i+1})) & \subseteq T_{\bar{P}_{i+1}}(M) \\
 & & = T_{\bar{P}_1}(M) \cup T_{\bar{P}_{i+1}}(M) \\
 & & = T_{\bar{P}_1}(M \cap U_{\bar{P}_i}) \cup T_{\bar{P}_{i+1}}(M) \\
\text{(by (4))} & & = T_{\bar{P}_i}(M(\bar{P}_i)) \cup T_{\bar{P}_{i+1}}(M) \\
\text{(by ind. hyp., Lemma 2, Th. 7)} & & \subseteq M(\bar{P}_i) \cup T_{\bar{P}_{i+1}}(M) \\
\text{(by Lemma 2)} & & \subseteq M(\bar{P}_i) \cup M \\
\text{(by (4)).} & & = M.
\end{array}$$

Since M was arbitrary, by the definition of $M(\bar{P}_{i+1})$ this proves (2), which concludes the proof. ■

Note that the theorem does not hold when the assumption that M is supported in the definition of $M(\bar{P}_i)$ is dropped. Indeed, let P be $p \leftarrow \neg q$. Then P_1 is empty, so $U_{\bar{P}_1} = \emptyset$ and $M(\bar{P}_1) = M_1 = \emptyset$. On the other hand $M_2 = \{p\}$ whereas

$$\bigcap \{M : M \models \bar{P}_2 \text{ and } M \cap U_{\bar{P}_1} = \emptyset\} = \bigcap\{\{p\}, \{q\}\} = \emptyset.$$

An Elementary Interpreter

Our purpose in this and the next section is to study the foundations of a proof theory for logic programs with negation. The study begins with three deliberately naive intuitions about inference using the clauses of a program:

1. to prove a ground atom A find a ground instance $A \leftarrow L_1$ &...& L_n of a clause in the program and prove each of $L_1,\ldots,L_n$;
2. to prove $\neg A$ show there is no proof of A from the clauses of the program;
3. if there is a proof of A, then there is also a proof of A which does not, itself, depend (recursively) on a proof of A.

A little reflection should suggest that, while (3) is correct, for programs with negation (1) leads to incompleteness and (2) is unsound.

Apart from the problematic aspects of (1) and (2) regarding completeness and soundness there are other more serious and surprising difficulties. Below, we introduce an interpreter which formalizes (1), (2), and (3) in a straightforward way. For certain programs we will see that there are multiple interpretations of the formal definition of the interpreter; in this sense the interpreter is ambiguous. The same difficulties arise with other definitions of provability in the case of logic programs with negation. In particular we find that the definition of SLDNF-resolution given in Lloyd [1984, p. 76] suffers from the same kind of ambiguity. Moreover, it is not at all clear that (1), (2), and (3) are even consistent in the case of some programs.

In this section, we show that the ambiguity in the definition of the interpreter vanishes when we restrict to stratified programs, and in the next section we show that the definition is, formally, consistent under this restriction. Thus for a stratified program P the definition of the interpreter uniquely specifies which ground atoms are provable from P. We also show that this need not be the case when P is not stratified. Moreover, when P is stratified, the set of ground atoms provable from P is M_P. Since the interpreter uniquely determines the class of ground atoms provable from P independently of how P is stratified, this result gives yet another proof that M_P is independent of the stratification.

Typically, there are ground atoms that are true in M_P but are not logical consequences of P. Thus, from the point of view of first-order logic our proof procedure given by the interpreter is unsound. However, soundness and completeness notions depend on what concept of logical consequence is being considered. For example, there are true sentences in number theory that are not (first-order) logical consequences of Peano's axioms. An enriched proof procedure that, starting from Peano's axioms, would allow us to prove some of these sentences must necessarily be unsound from the point of view of ordinary first-order logic, but should certainly not be dismissed because of this. The point is that notions of logical consequence can be based on *intended* models rather than on *all* models.

This is the attitude we take here: A stratified program P has a unique intended model, namely M_P, and we reduce the notion of logical consequence from P to that of truth in M_P.

The interpreter uses ground instantiations of the clauses of the program. We do not claim such an interpreter is adequate for real programming, but it does have two important properties. First, it is an analytical tool for investigating the foundations of a proof theory for logic programs with negation. Second, it is easy to see how to implement it as an executable Prolog program that gives us an immediate extension from ground instances of clauses.

We give a (for the moment informal) example that illustrates what should be taken as a proof based on (1), (2), and (3).

EXAMPLE 5

Consider a program P:

$$p \leftarrow q \,\&\, \neg r,$$

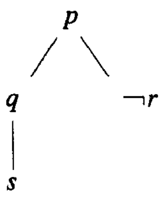
$$q \leftarrow p,$$

$$q \leftarrow s,$$

$$r \leftarrow t,$$

$$s$$

and the tree T:



(Note that we are using T to stand for an operator, and T to stand for a tree. It will also be clear from context which is meant.) Our intention is to regard T as proving p from P in the following way: s is a fact in P, so P immediately proves s, hence q. Now, there is no proof tree for t, hence no proof tree for r. This establishes $\neg r$, and consequently p. The interpreter must also "trap" loops. In this example it would be useless to try to prove q by proving p. ■

We now formalize these notions. We first define the class of objects that will be used by the interpreter to construct proof trees.

DEFINITION 14

Let U be the Herbrand base of a language L. An *implication tree* over U is inductively defined by the following rules.

1. For each $A \in U$, A and $\neg A$ are implication trees over U.
2. If $T_1,\ldots,T_n$ are (not necessarily distinct) implication trees, and $A \in U$, then

$$A(T_1,\ldots,T_n)$$

is an implication tree over U. We may also denote this tree by

$$A \leftarrow T_1 \,\&\, \ldots \,\&\, T_n. \ ■$$

When the context is clear we may sometimes omit the phrase "over U." An implication tree is identifiable with a (graphical) tree in the obvious way where we assume multiple copies of the same literal are distinguishable.

We will now introduce a sequence of restrictions on implication trees that will bring us to those trees that our interpreter can return as proof trees. Informally, it should be evident that along any path on a proof tree, the same ground literal need not occur more than once. Accordingly, we have the following definition.

DEFINITION 15
An implication tree T is *loop-free* iff T contains no path with two distinct occurrences of a node labelled A, for any $A \in U$. ■

Note that negative literals can only occur as leaves.

EXAMPLE 6
The implication tree given below is loop-free, and, we will see, proves q from

$q \leftarrow \neg q.$

$$\begin{tikzcd}
q \arrow[d, dash] \\
\neg q
\end{tikzcd}$$

The class of implication trees over U of course ignores the structure of logic programs whose Herbrand bases are contained in U. Because we want to use implication trees to prove the consequences of logic programs, we must select those implication trees that can serve as proofs. We shall use the term "compatible" to link implication trees with programs in this way. Proposition 1 makes precise what it is that compatible implication trees do prove. ■

DEFINITION 16
Let T be an implication tree, and P a logic program.

1. If T is $\neg A$, and A is ground, then T is *compatible* with P.
2. If T is A, and A is ground, then T is *compatible* with P.
3. If T is $A(\mathrm{T}_1,\ldots,\mathrm{T}_n)$, T_i is *compatible* with P $(i = 1,\ldots,n)$, $B_1,\ldots,B_n$ are the roots of $\mathrm{T}_1,\ldots,\mathrm{T}_n$ respectively, and

   $A \leftarrow B_1\&\ldots\&B_n$

is in ground (P), then T is compatible with P. Here ground (P) stands for the set of all closed instances of clauses from P. ■

The next definition allows the statement of Proposition 1 to be more succinct. Here and elsewhere, the symbol $\models$ is used to mean "first-order" logical provability.

DEFINITION 17

Let T be an implication tree compatible with P. A negative leaf $\neg A$ is *loop-trapped* if A occurs on the path from the root of T to $\neg A$. ■

PROPOSITION 1

Let T be a loop-free implication tree compatible with P, and let A be the root of T. Suppose $\neg B_1,\ldots,\neg B_k$ are all and only the negative leaves of T that are *not* loop-trapped. Then

$$P \models A \vee B_1 \vee \ldots \vee B_k.$$

Proof: Straightforward by induction on the height of T. ■

COROLLARY 3

Let T be a loop-free implication tree compatible with P in which every negative leaf is loop-trapped. Then

$$P \models \text{root(T)} \quad ■$$

In general, Proposition 1 and its corollary (fortunately!) do not have natural converses. That is, for a program that uses negation in a nontrivial way, a compatible implication tree may prove its root although not every negative leaf is loop-trapped. Consider the following example. (With the exception of Proposition 1, we are still being informal about "proof.")

EXAMPLE 7

P_1: $p \leftarrow q \& r,$

$q \leftarrow p,$

$q \leftarrow \neg s,$

$s \leftarrow p,$

$r.$

P_2: $p \leftarrow \neg q.$

First, in the case of program P_2

$$\begin{tikzcd} p \arrow[d, no head] \\ \neg q \end{tikzcd}$$

should indeed prove p since there is no proof of q. It is instructive to explicitly see the considerations in building an implication tree to prove p using P_1.

Step 1:

$$\begin{tikzcd} & p \arrow[dl, no head] \arrow[dr, no head] & \\ q & & r \end{tikzcd}$$

is an initial segment of an implication tree. r is a positive leaf that is also a fact in P_1. Thus, r is proved.

Step 2: There are two possibilities for extending the initial segment of Step 1.

(a)

$$\begin{tikzcd} & p \arrow[dl, no head] \arrow[dr, no head] & \\ q \arrow[d, no head] & & r \\ p & & \end{tikzcd}$$

(b)

$$\begin{tikzcd} & p \arrow[dl, no head] \arrow[dr, no head] & \\ q \arrow[d, no head] & & r \\ \neg s & & \end{tikzcd}$$

Alternative (a) is not an initial segment of any *loop-free* implication tree. Hence, there is no need to bother trying to prove p using alternative (a). Now, to use (b) we need to show that there is no proof of s. If there were, then we would have a proof of p and at the same time know that the alternative (b) fails to yield a proof. But (a) or (b) are, intuitively, the only ways of proving p. Thus, there is no proof of s, so (b) serves to prove p. We stress here that we are only giving evidence for what should be regarded as a proof, and are not trying to be rigorous. ■

We now define our interpreter. Let $\bar{U} = U \cup \neg U$, where $\neg U = \{\neg A \mid A \in U\}$.

DEFINITION 18

Let P be a logic program with Herbrand base U, and let IT be the set of implication trees over U. Then $I_p \subseteq \bar{U} \times \text{IT} \times 2^U$ is defined by

1. $I_P(A,A,S) \Longleftrightarrow A$ is in ground (P), and $A \notin S$.
2. $I_P(A,A \leftarrow T_1\&\ldots\&T_n,S) \Longleftrightarrow A \notin S$, and for some $A \leftarrow B_1\&\ldots\&B_n \notin$ ground (P) $I_P(B_1, T_1, \{A\} \cup S)$ and...and $I_P(B_n, T_n, \{A\} \cup S)$
3. $I_P(\neg A, \neg A, S) \Longleftrightarrow$ there does not exist a T such that $I_P(A, T, S)$.
4. Not $I_P(L,T,S)$ whenever $I_P(L,T,S)$ is not in any of the forms given in (1), (2), (3). ■

The reader should verify that the interpreter constructs the proof tree for p using P_1 as outlined in the previous example. The third argument of I_P carries information for the loop trap.

DEFINITION 19
T is a *proof tree* (w.r.t.P) iff $\exists A[I_P(A, T, \phi)]$. If T is a proof tree with root A, we say that T proves A. ■

The definition given for I_P is both ambiguous and computationally ineffective if P is left unconstrained. We shall see that the ambiguity vanishes if P is stratified. The noncomputability vanishes under additional constraints. In particular, if P is stratified, then any constraint on P that yields a decidable standard interpretation will result in I_P itself being a decidable relation.

The ambiguity in the definition of I_P lies in the fact that, in general, there is more than one relation on $\bar{U} \times \text{IT} \times 2^U$ that satisfies the definition. We shall demonstrate this difficulty with an example, then prove that the ambiguity essentially vanishes if P is stratified. In the following section we shall show that an unambiguous "bottom-up" inductive definition can be given that defines a unique relation on $\bar{U} \times \text{IT} \times 2^U$ that satisfies the definition of I_P.

EXAMPLE 8
Consider the following program P that is not stratified.

$q(0)$,

$p(x) \leftarrow \neg p(s(x))$.

To simplify the notation, let n abbreviate

$$\underbrace{s(s(\ldots s}_{n \text{ times}}(0)\ldots))$$

Applying the interpreter to $p(0)$ we have

$$\begin{tikzcd}[row sep=small] I_P(p(0), p(0), \phi) \arrow[d, dash] & \text{iff } I_P(\neg p(1), \neg p(1), \{p(0)\}). \\ \neg p(1) & \\ & \text{iff not } \exists T[I_P(p(1), T, \{p(0)\})] \end{tikzcd}$$

We therefore have:

$$\begin{tikzcd}[row sep=small] p(0) \arrow[d, dash] & \text{proves } p(0) \text{ iff} & p(1) \arrow[d, dash] & \text{is not a proof tree} \\ \neg p(1) & & \neg p(2) & \\ & \text{iff} & p(2) \arrow[d, dash] & \text{proves } p(2) \\ & & \neg p(3) & \\ & \text{iff} & p(3) \arrow[d, dash] & \text{is not a proof tree} \\ & & \neg p(4) & \\ & \text{iff} \ldots . & & \end{tikzcd}$$

Since this chain of equivalences is not terminating, we may suppose that every assertion in the chain is true, or that every assertion in the chain is false. In either case we satisfy the definition of I_P. ■

EXAMPLE 9

The interpreter that we have given is, in general, meaningful as a terminating *procedure* only in very controlled circumstances; for instance, for function symbol free programs. To see nonterminating behavior on a program with function symbols, consider the stratified program P with the following clauses:

$$p(x) \leftarrow p(s(x)),$$
$$q \leftarrow \neg p(0).$$

In seeking to prove q, I_P costructs

$$\begin{tikzcd}[row sep=small] q \arrow[d, dash] \\ \neg p(0) \end{tikzcd}$$

In seeking to show that there is no proof tree for $p(0)$, I_P constructs the sequence of proof trees

$$\begin{tikzcd}
p(0) \arrow[d, dash] & p(0) \arrow[d, dash] & p(0)\ldots. \arrow[d, dash] \\
p(s(0)) & p(s(0)) \arrow[d, dash] & p(s(0)) \arrow[d, dash] \\
 & p(s(s(0))) & p(s(s(0))) \arrow[d, dash] \\
 & & p(s(s(s(0))))
\end{tikzcd}$$

However, the unique relation on $\bar{U}_P \times \mathrm{IT}_P \times 2^{U_P}$ that satisfies the definition of I_P is

$$\{< q, \begin{array}{c} q \\ | \\ \neg p(0) \end{array}, S > | S \in 2^{U - \{q\}}\}. \blacksquare$$

Next we show that the ambiguity in the definition of I_P essentially vanishes when P is stratified. By "essentially," we mean that when P is stratified, and if R is defined by

$$R(A,S) \Longleftrightarrow \exists \mathrm{T}\, [I_P(A,\mathrm{T},S)]$$

then R is uniquely determined. Now, as a matter of fact when P is stratified, the definition of I_P does uniquely determine I_P on $\bar{U} \times \mathrm{IT} \times 2^U$, but it is convenient to defer the proof of this fact until after introducing a "bottom-up" definition for an interpretation of I_P in the next section.

Next we prove a technical lemma about loop-trapping in our interpreter. Note that the interpreter will not apply any rule to a goal A if A is in S. Normally, S is the set of proper ancestors of A. Now suppose that A is defined in stratum k of a program P. Intuitively, our lemma says that we can add to S all ground literals defined in the strata above k, without changing the behavior of the interpreter.

LEMMA 14

Let P be stratified by $P_1 \dot{\cup} \ldots \dot{\cup} P_n$. Let $V_j = Def_{Q_j}$ where $Q_j = P_j \cup \ldots \cup P_n$ ($j = 1,\ldots,n$), and let $V_{n+1} = \phi$. For $j = 2,\ldots,n+1$, $S \subseteq V_j$ and T compatible with $P_1 \cup \ldots \cup P_{j-1}$ the following holds:

$I_P(\text{root}(T), T, \phi) \Longleftrightarrow I_P(\text{root}(T), T, S)$ (1)

Proof: Proceed by induction on j $(2 \leq j \leq n)$.

$j = 2$: P_1 is a positive program. Since T is compatible with P_1 and $S \subseteq V_2$, no node in T occurs in S. The equivalence (1) follows immediately by a straightforward induction using clauses (1) and (2) in the definition of I_P.

$j \Rightarrow j + 1$: For each implication tree T and node N in T let S_{T_N} be the set of nodes occurring on the path in T from the root to the parent of N. (In particular, $S_{T.\text{root}(T)} = \phi$.) Recall Proposition 1. We proceed by strong induction.

Let T be compatible with $P_1 \dot{\cup} \ldots \dot{\cup} P_j$. Then:

$I_P(\text{root}(T), T, \phi)$

iff (by clauses (1) and (2) in the definition of I_P)

$I_P(\neg B, \neg B, S_{T, \neg a})$ for every negative leaf $\neg B$ of T.

iff (by clause (3) in Definition 18)

not $I_P(B, T', S_{T, \neg B})$ for every negative leaf $\neg B$ of T, and implication tree T′

iff (since not $I_P(\text{root}(T'), T', X)$ when T′ is not compatible with P)

not $I_P(B, T', S_{T, \neg B})$ for every negative leaf $\neg B$ of T, and implication tree T′ compatible with $P_1 \cup \ldots \cup P_k$ where P_k is the stratum in which the relation symbol of B is defined. (Note that $k < j$.)

iff (by induction hypothesis, since $S_{T, \neg B} \subseteq V_{k+1}$)

not $I_P(B, T', \phi)$ for every B and T′ as in the previous step

iff (again, by induction hypothesis)

not $I_P(B, T', S \cup S_{T, \neg B})$ for every B, and T′ as in the previous step

iff (by clause (3) of Definition 18)

$I_P(\neg B, \neg B, S \cup S_{T,\neg B})$ for every negative leaf $\neg B$ of T

iff (by rules (1) and (2) of 18)

$I_P(\text{root(T)},T,S)$. (2) ■

We now give the theorem that, in effect, says the interpreter unambiguously "computes" the standard model M_P of P, when P is stratified. The statement of the theorem may at first seem a bit arcane, but recall that the definition of I_P does not in general uniquely determine a relation on $\bar{U} \times \text{IT} \times 2^U$. We are about to show, under the assumption that P is stratified, that *if* the definition of I_P is satisfied on $\bar{U} \times \text{IT} \times 2^U$, then I_P does indeed prove precisely those A that are true in the standard model of P. Proving that I_P can be satisfied at all is somewhat harder. This is in part the point of the following section.

THEOREM 13

Let P be stratified. Suppose there is an interpretation of I_P on $\bar{U} \times \text{IT} \times 2^U$ that satisfies the definition of I_P. Under such an interpretation we have the following equivalences for all A in U_P:

1. $\exists T[I_P(A,T,\phi)]$ iff $A \in M_P$.
2. $I_P(\neg A, \neg A, \phi)$ iff $A \notin M_P$.

Proof: P is stratified by $P_1 \dot{\cup} P_2 \dot{\cup} \ldots \dot{\cup} P_n$, for some $P_1,\ldots,P_n$. For each A in U, let r_A be the relation symbol occurring in A. If r_A is a relation symbol in the language in which P is a set of formulas (recall that the language with which P is associated may be larger than the language generated by symbols occurring in P), but r_A does not occur in P, then, by definiton, we shall say r_A is defined in P_1. Otherwise, r_A is defined in one of the P_i.

We proceed by induction on the index j of the stratum in which r_A is defined.

$j = 1$. $A \in M_P$ iff $A \in M_1$. The following equivalence holds:

$A \in T_{P1}\uparrow n(\phi)$ iff

there is a loop-free implication tree T compatible with P such that height (T) $\leq n$ and $I_P(A,T,\phi)$.

This equivalence follows by an easy induction on n. $M_1 = \bigcup_{n=0}^{\infty} T_{P_1}\uparrow n(\phi)$.

Thus, (1) follows. (2) follows from (1), clause (3) of (18), and that all clauses about r_A are in P_1.

$j \Rightarrow j-1$. The induction assumption we take is the following assertion. For every A such that r_A is defined in some stratum indexed by $j' \leq j$:

$$\exists \mathrm{T}[I_P(A,\mathrm{T},\phi)] \text{ iff } A \in M_j', \qquad \text{and } I_P(\neg A,\neg A,\phi) \text{ iff } A \notin M_j'. \tag{3}$$

To complete the proof it suffices to show that

For every A such that r_A is defined in P_{j+1}:

$$\exists \mathrm{T}[I_P(A,\mathrm{T},\phi)] \text{ iff } A \in M_{j+1} \qquad \text{and } I_P(\neg A,\neg A,\phi) \text{ iff } A \notin M_{j=1}. \tag{4}$$

Let $A \in U$ such that r_A is defined in P_{j+1}. If T is an implication tree that is not compatible with P_{j+1}, or $A \neq$ root(T), then not $I_P(A,\mathrm{T},X)$ for any $X \subseteq 2^U$, since it is easy to see that $I_P(A,\mathrm{T},X)$ is false if T is not compatible with P. Let T be compatible with P_{j+1}, and let A = root(T). Then:

$I_P(A,\mathrm{T},\phi)$

iff (by clause (1) and (2) of Definition 18)

$I_P(\neg B,\neg B,S_{\mathrm{T},\neg B})$ for every negative leaf $\neg B$ of T

iff (by clause (3) of Definition 18)

not $I_P(B,\mathrm{T}',S_{\mathrm{T},\neg B})$ for every negative leaf $\neg B$ of T, and implication tree T′

iff (by Lemma 14)

not $I_P(B,\mathrm{T}',\phi)$ for every negative leaf B of T and implication tree T′

iff (by clause (3) of Definition 18)

$I_P(\neg B,\neg B,\phi)$ for every negative leaf B of T

iff (by the induction hypothesis)

$B \notin M_j'$ for every negative leaf B of T, where j' is the index of the relation symbol of B

So we have

$$I_P(A,\mathrm{T},\phi) \text{ iff} \qquad (5)$$
$$B \notin M_{j+1} \text{ for every negative leaf } B \text{ of T}$$

Now, suppose, $I_P(A,\mathrm{T},\phi)$. By Proposition 1,

$$P_{j+1} \models A \vee B_1 \vee \ldots \vee B_k \qquad (6)$$

where $\neg B_i, \ldots, \neg B_k$ are all and only the negative leaves of T.

$B_i \notin M_{j+1}$ $(i = 1,\ldots,k)$ by (5),

$M_{j+1} \models A \vee B_1 \vee \ldots \vee B_k$ by Theorem 7 and (6).

Therefore $A \in M_{j+1}$.

Conversely, suppose $A \in M_{j+1}$. Then $A \in T_{P_{j+1}} \uparrow m(M_1)$, for some finite m. A routine induction argument shows that there is an implication tree T (of height $\leq m$) that is compatible with P_{j+1}, has A as its root, and has the property that $B \notin M_j$ for every negative leaf B. Thus, $I_P(A,\mathrm{T},\phi)$. This completes the proof of the theorem. ■

Existence of the Interpreter

In this section we prove that an interpreter satisfying Definition 18 exists when P is a stratified program.

Theorem 13 shows that if the definition if I_P is satisfiable over $\bar{U}_P \times \mathrm{IT} \times 2^{U_P}$, then the set of triples,

$$\{<A, \mathrm{T}, \emptyset>\}$$

in the satisfying relation is uniquely determined. It remains to show that the definition of I_P is, indeed, satisfiable over $\bar{U}_P \times \mathrm{IT} \times 2^{U_P}$. The proof of Theorem 13 is almost sufficient for this purpose. What is lacking is a definition D of a relation R over $\bar{U}_P \times \mathrm{IT} \times 2^{U_P}$ for which D is manifestly uniquely satisfiable and for which R satisfies the definition of I_P. For this purpose it suffices to give D as a "bottom-up" inductive definition.

Suppose that D has been given for R so that

$$\exists \mathrm{T}[R(A,\mathrm{T},\emptyset)] \Rightarrow A \in M_P.$$

R itself then serves an an interpreter for stratified programs, and the difficulties concerning the ambiguity inherent in the definition of I_P vanish. Therefore, the objection may be raised that the definition of I_P may be bypassed in the development of our analysis of stratified programs. That is, why should we present two interpreters that are extensionally the same? The objection can be dispelled on two grounds. First, I_P is a "top-down," recursive, backward-chaining interpreter, and it will be seen that R is "bottom-up," inductive, and forward-chaining. Both points of view are independently interesting. Secondly, once we know that I_P presents no logical difficulties in the context of stratified programs, the definition of I_P is concise and simple. We now turn to the construction of R and prove that R satisfies the definition of I_P.

EXAMPLE 10
Let P be

$$q,$$

$$p \leftarrow \neg q.$$

Now,

$$I_P(p, p \leftarrow \neg q, \{q\})$$

follows from the definition of I_P, yet $p \notin M_P$. ■

Example 10 illustrates that we can prove undesired consequences when starting from nonempty loop traps. The following construction circumvents this difficulty.

DEFINITION 20
If Q is obtained from P by the following construction, we say that Q *is the result of filtering* P through S.

Let L be a first-order language. U the Herbrand base of L, and let P be a stratified program in the language L with stratification given by

$$P = P_1 \dot{\cup} \ldots \dot{\cup} P_n$$

Let

$$P' = P'_1 \dot{\cup} \ldots \dot{\cup} P'_n$$

be the collection of ground instances of clauses in P.

Suppose $S \subseteq U$. Obtain

$$Q = Q_1 \dot{\cup} \dots \dot{\cup} Q_n$$

from P' by removing all clauses with heads in S. ■

DEFINITION 21

Let L be a first-order language with Herbrand base U. Let P be a stratified program in the language L with stratification given by

$$P = P_1 \dot{\cup} \dots \dot{\cup} P_n$$

Let $S \subseteq U$, and let

$$Q^S = Q_1^S \dot{\cup} \dots \dot{\cup} Q_n^S$$

be the result of filtering P through S.

Let $A \in U$. Define

$$\text{index}\ (A) = \begin{cases} 1, \text{ if the relation symbol of } A \text{ is not defined in any stratum of } Q^S, \\ \\ k, \text{ if the relation symbol of } A \text{ is defined in stratum } Q_k^S \end{cases}$$

(Note that Q^S is in general a stratified program consisting of infinitely many ground clauses.) Let $M_1,\dots,M_n$ be the models of $Q_1^S,\dots,Q_n^S$ respectively where M_i is obtained from iter $(T_{Q_1^s},\dots,T_{Q_1^s}, \phi)$, (thus $M_{Q^S} = M_n$.) Let

T be *proof tree with respect to S and P* iff

T is a loop-free implication tree compatible with Q^S, has positive root, and for every negative leaf $\neg B$ of T, $B \notin M_{\text{index}(B)}$.

Finally, let R be the relation of $\bar{U} \times \text{IT} \times 2^U$ defined by

$R(A,\text{T},S) \iff$ root (T) $= A$ and T is a proof tree with respect to S and P

and

$R(\neg A, \neg A,S) \iff$ not $R(A,\text{T},S)$ for every implication tree T. ■

COMMENT

It was claimed above that R would be given as a forward chaining interpreter. Note that R depends on Definition 20 which, in turn, depends on the $Q_1,\dots,Q_n$.

In effect, we could use Definition 20 to construct proof trees in a "bottom-up" inductive way using $T_{Q_1},\dots,T_{Q_n}$.

THEOREM 14
R satisfies the definition of I_P.

Proof: The proof is not deep, but for the sake of exposition of what is at stake we nonetheless give it.

Suppose that P, L, U, and S are as in Definition 21. Let $A \in U$, and let T be an implication tree with respect to U. To show that R satisfies the definition of I_P, we must show

$$R(A,\mathrm{T},S) \qquad (1)$$

$$\Leftrightarrow$$

$$\mathrm{T} = A \text{ and } A \text{ is in ground}(P) \text{ and } A \notin S \qquad (2)$$

or

$$\mathrm{T} = A \leftarrow \mathrm{T}_1\&\dots\&\mathrm{T}_k \text{ and } A \notin S \text{ and for some} \qquad (3)$$
$$(A \leftarrow B_1\&\dots\&B_k) \in \text{ground}(P):$$
$$R(B_1,\mathrm{T}_1,\{A\} \cup S) \text{ and}\dots\text{and } R(B_k,\mathrm{T}_k,\{A\} \cup S)$$

and

$$R(\neg A, \neg A,S) \Leftrightarrow \text{not } \exists \mathrm{T} \in \mathrm{IT} \text{ such that } R(A,\mathrm{T},S). \qquad (4)$$

Now, (4) is just a restatement of the second part of the definition of R.

From Definition 21, to show equivalence (1) it suffices to show

$$(2) \text{ or } (3) \text{ iff } T \text{ is a proof tree with respect to } S \text{ and } P, \text{ and root (T)} = A \qquad (5)$$

Let Q be the result of filtering P through S. Suppose A is a head of a clause in Q. Let Q' be the result of filtering P through $\{A\} \cup S$. Q' does not differ from Q in strata below that stratum in which A occurs. Moreover, if A does not occur in T, then T is compatible with Q iff T is compatible with Q'.

Let T be an implication tree with positive root over U. There are two cases.

Case 1: $\mathrm{T} = A$. Then T is a proof-tree with respect to S and P iff (2).

Case 2: $\mathrm{T} = A \leftarrow \mathrm{T}_1\&\dots\&\mathrm{T}_k$. Without loss of generality we may suppose $A \in S$ and that $A \leftarrow \text{root}(\mathrm{T}_1)\&\dots\&\ \text{root}(\mathrm{T}_k) \in \text{ground}(P)$, else both sides of (5) are false.

T is a proof tree w.r.t. S and P

iff (by Definition 21)

T is a loop-free implication tree compatible with Q and for every negative leaf $\neg B$ of T, $B \notin M_{\text{index}(B)}$, where $M_1, \ldots M_n$ are the standard interpretations of $\bar{Q}_1, \ldots, \bar{Q}_n$ respectively, where $\bar{Q}_i$ denotes $Q_1 \cup \ldots \cup Q_i$

iff (since $B \in M_{\text{index}(B)}$, iff there is a proof tree w.r.t. S and Q with root B, see below) (6)

for each T_i, $i \in \{1, \ldots, k\}$, if root(T_i) is positive, then T_i is a loop-free implication tree compatible with Q, and for every negative leaf $\neg B$ of T_i,$B \notin M_{\text{index}(B)}$,

and

if root(T_i)is negative, say $\neg C_i$>, then there is no proof tree with respect to S and Q with root C_1

iff (since Q and Q' do not differ in strata below that in which A is defined)

for each T_i, $i \in \{1, \ldots, k\}$ if root (T_i) is positive, then T_i, is a loop-free implication tree compatible with Q', and for every negative leaf $\neg B$ of T_i, $B \notin M'_{\text{index(B)}}$, (where M'_j is the standard interpretation of Q'_j,$j = 1, \ldots, k$)

and

if root (T_i) is negative, say $\neg C_i$, then there is no proof tree w.r.t. $\{A\} \cup S$ and Q' with root C_i, where Q' is the result of filtering P through $\{A\} \cup S$

iff

(3) where $B_i = \text{root}(T_i)$, $i \notin \{1, \ldots, k\}$.

Equivalence (6) was justified by

$B \in M_{\text{index}(B)}$ iff there is a proof tree w.r.t. S and Q with root B.

Proof trees with respect to S and Q are loop-free implication trees compatible with S and Q because Q contains no clause head in S. Thus the "if direction" of (6) follows from Proposition 1.

Suppose $B \in M_{\text{index}(B)}$. It is easy to show by an inductive construction that there is a loop free implication tree T compatible with Q for which every negative leaf $\neg C$ has the property that $C \notin M_{\text{index}(C)}$. Once again, since no clause head in Q occurs in S, T is a proof tree w.r.t. S and Q. ■

COROLLARY 4

If P is a stratified program, then M_P, the standard model of P, is independent of the stratification of P.

Proof: The corollary follows directly from the statement of Theorem 13 since we have proved that the definition of I_P is satisfiable over $\bar{U} \times \text{IT} \times 2^U$. ■

We conclude this section with a remark on the computational complexity of I_P. We showed in the previous section that in general I_P by itself does not yield a computation procedure capable of verifying that $A \in M_P$ even when M_P and all the lower stage standard interpretations are uniformly decidable. Nevertheless one of our principal areas of application is that of stratified programs without function symbols. For such programs I_P does indeed determine a useful computation procedure, since the Herbrand base of P is finite. From the point of view of worst case complexity, I_P is no worse than ordinary depth first, SLD-resolution-based interpreters for purely positive programs.

EXAMPLE 11

Let P_3 consist of

$c_3(0,0,0)$,

$c_3(X,Y,1) \leftarrow c_3(X,Y,0)$,

$c_3(X,1,0) \leftarrow c_3(X,0,1)$,

$c_3(1,0,0) \leftarrow c_3(0,1,1)$.

The goal $(\leftarrow c_3(1,1,1))$ produces a successful SLD path consisting of eight nodes. Similarly, I_{P_3} constructs a proof tree, with $c_3(1,1,1)$ as root of height eight. P_3 causes I_{P_3} as well as SLD-resolution to, in effect, run through a count-down of a 3-bit binary counter. It should be clear how to construct P_n from example P_3. ■

Now, let P be a positive program in which no function symbols other than constants occur. Suppose P contains r relation symbols, each with arity $\leq a$, and c constants. Let U be the Herbrand base of P. Then

$$\text{size}(U) \leq rc^a,$$

Letting $|| \, T_P \, ||$ = least n such that $T_P \uparrow (n-1)(\phi) = T_P \uparrow n(\phi)$, we have

$$|| \, T_P \, || \leq \text{size}(U).$$

For the programs P_n of example

$$|| \; T_{P_n} \; || = \text{size}(U) = 2^n$$

and both I_{P_n} as well as SLD-resolution are forced to, in effect, enumerate all of U when starting with ($\leftarrow c_n(1,\ldots,1)$) as goal. Now, size($P_n$) = $O(n)$. It follows that I_P as well as SLD-resolution require, in the worst case, $O(2^{size(P)})$ steps to succeed. Lastly, note that in our interpreter the loop check need only add a linear time complexity component, because we can represent S as an ordered tree; each path in the tree represents a literal A. Lookup and insertion time of A is $O(length(A))$.

Other Views of Negation and Stratified Programs

Our way of interpreting negation in the case of stratified programs is through choosing M_P as the set of true facts about P. It is helpful to relate this interpretation of negation with two other ones proposed in the literature.

Closed World Assumption (CWA)

Reiter [1978] proposed the *closed world assumption* (CWA) as a way of adding negative information to logic programs. According to this view any (atomic) fact which does not follow from a given program is assumed to be false. Thus by definition

$$CWA = P \cup \{\neg A : A \textit{ is a ground atom and not } P \models A\}.$$

where "$\models$": stands for provability in first order logic. (Reiter disallowed function symbols but the problems discussed here do not depend on this assumption.)

This view is certainly a natural one when the program is just a collection of facts—if manager(Jones) is not in our database, then we usually conclude that Jones is not a manager. Moreover, Reiter [1978] proved that in the case of positive programs, CWA is consistent. But while trying to extend this view to arbitrary programs somehow our intuitions get lost and, we encounter difficulties (of which Reiter was perfectly aware). Restriction to stratified programs is no help. Consider, for example, the stratified program P: $p \leftarrow \neg q$. Then P is semantically equivalent to $p \vee q$, which implies neither p nor q. Thus, $CWA = P \cup \{ \neg p, \neg q\}$ and is inconsistent. The closed world assumption leads to difficulties here. Negation seems to be too strong when interpreted through CWA: Any uncertainty is resolved in a negative way.

Completions of Programs

Another way of adding negative information to the program is that proposed by Clark [1978] and called the *completion* of a program. His idea was to reinterpret the implications within the program as equivalences. In this way one adds to the program the "only if" part which allows us to infer negative consequences.

More formally the *completion* is defined as follows. (We slightly depart here from the original definition as we omit the equality axioms which are automatically satisified in Herbrand models when "=" is interpreted as identity.)

Let $x_1,\ldots,x_k$, be some variables not appearing in the program. First, transform each clause

$$p(t_1,\ldots,t_k) \leftarrow L_1\&\ldots\&L_m$$

of P into

$$p(x_1,\ldots,x_k) \leftarrow \exists y_1,\ldots,y_i(x_1 = t_1)\&\ldots\&(x_k = t_k)\&L_1\&\ldots\&L_m$$

where $y_1,\ldots,y_i$ are the variables of the original clause.

Next, change each set of the transformed clauses of the form

$$p(x_1,\ldots,x_k) \leftarrow S_1,$$
$$\vdots$$
$$p(x_1,\ldots,x_k) \leftarrow S_n,$$

where $n \geq 1$, into

$$\forall x_1,\ldots,x_k p(x_1,\ldots,x_k) \leftrightarrow S_1 \vee \ldots \vee S_n.$$

Then we denote by *comp(P)*, the *completion* of P, the formula consisting of the conjunction of these equivalences and of formulas

$$\forall x_1,\ldots,x_k \neg q(x_1,\ldots,x_k)$$

for each relation q which appears in the program but not in a head of a clause.

We wish to interpret "=" as identity. Therefore, we add the following clause to the definition of semantics: For two variable-free terms *s.t.* $s = t$ is true in I iff s and t are identical. It is well known (see e.g., Shepherdson [1984]) that for positive programs comp(P) is consistent. We now prove that for stratified programs comp(P) is consistent as well.

In Apt and van Emden [1982] models of comp(P) for a positive program P were characterized as fixed points of T_P. Fortunately, this characterization remains true in the presence of negation. We have

THEOREM 15
Let P be a program. Then I is a model of comp(P) iff $T_P(I) = I$.

Proof: The definition of comp(P) we use is slightly different than those given in Apt and van Emden [1982] (called there the IFF definition) or Lloyd [1984]. Nevertheless, all steps of the (straightforward) proof remain the same. A doubting reader is encouraged to check the proof sketched in Lloyd [1984]. ■

This immediately implies the following theorem.

THEOREM 16
Let P be a stratified program. Then comp(P) is consistent.

Proof: We exhibited in the section on model theory a fixed point of T_P—which is the standard model M_P. Thus, M_P is a model of comp(P) which proves the consistency of comp(P). ■

It is important to note that the views of negation represented by comp(P) and M_P do not coincide. Indeed, consider the stratified program P consisting of

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

Then comp(P) is

$$(p \leftrightarrow p)\ \&\ (q \leftrightarrow \neg p)$$

which is equivalent to $q \leftrightarrow \neg p$. We thus have that it is not the case that *comp*(P) $\models \neg p$ whereas $M_P = \{q\}$. Hence $M_P \models \neg p$.

A Discussion

It is useful to have a closer look at the consequences of the fact that we interpret negation by model theoretic means. This implies that each atomic fact about a stratified program P is considered either true (when it belongs to the model M_P) or false (when it does not belong to the model M_P).

This duality does not need to take place when negation is interpreted through proof theoretic means. A common feature of the Clark [1978] and Reiter [1978] approaches to negation is to extend a given program P to a theory, say N_P, in which no new atomic facts can be proved and interpret negation by means of that theory. That is to say, for an atomic fact A

A is true iff $N_P \models A$

A is false iff $N_P \models \neg A$

Now, if N_P is not a complete theory, there is an atomic fact A which is neither true nor false. (We call here a theory T *complete* if for each atomic fact A either $T \models A$ or $T \models \neg A$ but not both.) When N_P is not complete, it is possible to establish a result of the form

A is true in all models of N_P iff A can be computed from P,

A is false in all models of N_P iff $\neg A$ can be computed from P.

In fact, when N_P is comp(P) this is the essence of the "completeness of the negation as failure" result proved in Jaffar, Lassez, and Lloyd [1983].

There, P is a positive program, N_P is comp(P), and the computation mechanism is SLD resolution with negation as failure. More precisely, we have for all atoms A

comp(P) $\models A$ iff $\leftarrow A$ can be refuted from P,

comp(P) $\models \neg A$ iff $\leftarrow \neg A$ can be refuted from P.

(The first line in effect states the completeness of the SLD-resolution originally proved by Hill [1974].) The case when neither comp(P) $\models A$ nor comp(P) $\models \neg A$ is simply not handled: The refutation process then leads neither to success nor to a finite failure, hence it always diverges. Thus, for such A not truth value of A w.r.t. comp(P) is not defined. In such cases, of course, there will be models M_1, M_2 of comp(P) such that $M_1 \models A$, and $M_2 \models \neg A$.

The situation changes when N_P is a complete theory. Then without any restrictions on the syntax we cannot obtain a completeness result *even* in the case of positive programs. Indeed, suppose otherwise. Let W be a recursively enumerable, non-recursive set of natural numbers. By the result of Tärnlund [1977] there exists a positive program P such that for some relation p for all n

$P \models p(n)$ iff $n \in W$.

Since N_P is complete, we have for all n

$N_P \models p(n)$ iff $n \in W$,

$N_P \models \neg p(n)$ iff $n \notin W$.

Indeed, if $n \notin W$, then $P \models p(n)$ so $N_P \models p(n)$ since N_P extends P. And if $n \notin W$, then not $P \models p(n)$ so not $N_P \models p(n)$, since in N_P no new positive facts can be proved. This proves the first line and the second follows by the completeness of N_P.

This has two consequences. First is that the provability in N_P is not recursively enumerable (by the second equivalence) so N_P cannot be recursively axiomatized (see Rogers [1967]). Second, no completeness result for effective computation mechanisms is possible. Indeed, in case of completeness we have for all n

$$\begin{aligned} n \notin W &\text{ iff } N_P \models \neg p(n) \\ &\text{ iff } p(n) \text{ is false} \\ &\text{ iff } \neg p(n) \text{ can be computed from } P \end{aligned}$$

so the relation "A can be computed from P" is not recursively enumerable, that is, the computation mechanism is not effective.

An example of an interpretation of negation through a complete theory N_P is Reiter's closed world assumption for positive programs. Indeed, as mentioned in the subsection on the CWA, it is consistent for positive programs, and if not $(P + CWA) \models A$, then not $P \models A$ and consequently $(P + CWA) \models \neg A$. By the above remarks CWA cannot be recursively axiomatized and no completeness result for an effective computation mechanism is possible.

The same negative results hold for our interpretation of negation through the model M_P which acts as a complete theory. Thus, we define

$$\begin{aligned} &A \text{ is true iff } M_P \models A \\ &A \text{ is false iff not } M_P \models A \end{aligned}$$

In general, while it is beyond the scope of this paper to prove it, it can be seen that for a stratified program P containing function symbols and having n strata, M_P can be non-recursively enumerable (in fact Σ_n^0-complete). The proof of this result is based on recursion-theoretic considerations similar to that of Blair [1982]. Thus, no effective procedure for computing M_P in general is possible.

The above discussion does not exclude a completeness result for stratified programs and the negation interpreted through Clark's completed database. We established the first, necessary, step by showing that for a stratified program P, comp(P) is consistent but we did not explore the issue any further. A warning should be issued to those wishing to investigate this problem. The following examples show that no completeness result w.r.t. comp(P) for SLD-resolution with negation as failure (called the SLDNF-resolution) or for our interpreter is possible.

EXAMPLE 12
Let P be the following program:

$p \leftarrow p,$

$q \leftarrow p,$

$q \leftarrow \neg p.$

Then P is stratified and comp(P) is equivalent to $(q \leftrightarrow p \vee \neg p)$, so *comp*($P$) $\models q$. On the other hand, there is no refutation of $\leftarrow q$ using SLDNF-resolution.

EXAMPLE 13
Let P be the following stratified program:

$p \leftarrow p,$

$q \leftarrow \neg p.$

Then comp(P) is equivalent to $q \leftrightarrow \neg p$ so not *comp*(P) $\models q$. However, our interpreter computes q. ■

Thus, some other computation mechanisms have to be considered. On the other hand, we have the following conjecture.

DEFINITION 22
Let P be a program and p,q two relation symbols of P.

1. We say that p *depends positively* on q if p depends on q and in the dependency graph of P there is at least one path from p to q which contains exactly an even number (possibly 0) of negative edges.
2. We say the p *depends negatively* on q if p depends on q and in the dependency graph of P there is at least one path from p to q which contains exactly an odd number of negative edges.
3. We say that P is *strict* if for no relation symbols p and q of P, p depends both positively and negatively on q. ■

DEFINITION 23
Let P be a program. Given a clause of P let

X stand for the set of the variables occurring on its left hand side.

Y for the set of the variables occurring in a positive literal of the body and

Z for the set of the variables occurring in a negative literal of the body.

We say that P satisfies the *strong covering axiom* if for each of its clauses

$$X \subseteq Y$$

and

$$Z \subseteq Y$$

holds. ■

The first implication is called in Shepherdson [1984] the *covering axiom*. The strong covering axiom ensures that in the SLDNF-resolution only *ground* negative literals need be evaluated. We can now formulate our conjecture.

CONJECTURE 1

Let P be a strict stratified program which satisfies the strong covering axiom. Then for every ground literal L.

$$\text{comp}(P) \models L \text{ iff } \neg L \text{ can be refuted from } P \text{ by SLDNF-resolution.} \ ■$$

(*Note*: Conjecture 1 was recently proved by Cavedon and Lloyd. Subsequently Kunen showed that the result holds even for non-stratified programs.)

Related Work

Syntax

As already mentioned in the introduction, stratified programs form a simple generalization of the class of formulas C given in section 5 of Chandra and Harel [1985], in which negation is handled in a similar, stratified way. The difference lies in the way the strata are related—for technical reasons, in their paper this is accomplished by additional relations which take care of the calls involving negative literals. Also, our class of semi-positive programs coincides with their class H'_j but they are concerned with different issues than we are and concentrate on the subject of definability of database queries. In particular, they do not allow function symbols.

Stratified programs were also introduced and studied independently in a recent paper of Van Gelder [1988] and, in the context of databases, in Naqvi [1986]. A form of stratification for logic programming without negation was first introduced by Sebelik and Stepanek [1982]. Our definition of stratified programs is a generalization of the concept of the *hierarchical constraint* of Clark [1978] according to which program relations can be assigned to levels so that each relation is defined only in terms of relations from the lower levels. The hierarchical constraint, in contrast to the stratificiation condition, rules out recursive definitions. In fact if we remove negation from the language, then programs reduce to positive programs. If we remove recursion from the language, then they reduce to programs with the hierarchical constraint. The notion of stratified programs has been further generalized to *locally stratified programs* in Przymusinski [1988].

Lloyd and Topor [1985] give a theoretical basis for deductive databases using PROLOG as the query evaluator. They show in particular that SLDNF-resolution does not *flounder* (does not reach a goal that contains only non-ground negative literals) with general programs and goals provided the program and goal is *allowed.* The concept of being *allowed* as applied to clauses is equivalent to our *strong covering axiom.*

Semantics

The semantics of logic programs with negation based on fixed points is a generalization of the approach originating with van Emden and Kowalski [1976] and further explored in Apt and van Emden [1982]. Chandra and Harel [1985] provide in an informal way a semantics for their class C of programs which corresponds to our model M_P where the partition of P is into appropriately ordered clusters.

An early approach to provide meaning to logic programs with negation was given by Minker [1982] in the context of deductive databases. For this purpose, he introduced the concept of a generalized closed world assumption (GCWA). Its semantic characterization employs minimal models.

The notion of minimality arises in many other studies of nonmonotonic reasoning as well—see e.g., the notion of *circumscription* due to McCarthy [1980]. In fact, recent work of Lifschitz [1988] provides an alternative definition of the model M_P in terms of a circumscription. His approach leads to a simpler proof of Theorem 11.

The notion of a supported model has also been introduced independently in Bidoit and Hull [1986] and called there a *causal model.* Przymusinski [1988] introduced the concept of a *perfect model* of a database and related it to circumscription and semantics of stratified databases. Stratified databases were recently studied in Apt and Pugin [1987], Lloyd, Sonenberg, and Topor [1986], and Topor and Sonenberg [1988].

Completeness Results in the Presence of Negation

In Clark [1978], a completeness result for programs with the hierarchical constraint is sketched. It relates completed programs with SLD-resolution with negation as failure. A rigorous proof is given in Shepherdson [1984]. Another completeness result is that of Jaffar, Lassez and Lloyd [1983] mentioned in the previous section but only for positive programs. See also Aquilano et al. [1986] for a completeness result in the presence of negation and absence of divergence ensured by syntactic criteria. A recent paper of Shepherdson [1988] provides an extensive overview of the use of negation in logic programming.

Nonmonotonic Reasoning

An entire issue of the journal *Artificial Intelligence* [1980] has been published on the subject and a conference organized (Proceedings [1984]). Gabbay [1985] provides a useful discussion of the problem of when a reasoning method can be viewed as a nonmonotonic logic. Our approach based on search for fixed points of nonmonotonic operators is very similar in nature to the one recently proposed by Sandewall [1985].

Interpreters and Other Computation Mechanisms

Our treatment of a computation mechanism in the form of an interpreter relates to the approach taken by Brough and Walker [1984], who studied interpreters with various stopping criteria for positive database-like programs.

Barbuti and Martelli [1986] prove the completeness of SLDNF-resolution for a sizeable class of naturally occurring logic programs called *structured* programs. Structured programs form a set of stratified programs.

Recently, Fitting [1985] and Gallier and Raatz [1987] proposed alternative computation mechanisms for logic programming based on, respectively, a tableau method and an interpreter using graph reduction.

Use of Logic Programming for Expert Systems

Walker [1986a] described an implemented expert system shell in Walker [1986a], called Syllog, which is based on logic programming. Syllog contains an inference engine that computes with database-like programs with negation allowed.

One of the important aspects of expert systems is the ability to reason in terms of uncertainty. Recently van Emden [1986] extended the results of Apt and van Emden [1982] to the case when the facts and rules have some certainty factor associated with them. His work can be viewed as orthogonal to ours.

Acknowledgment

We thank Ashok Chandra for helpful comments on the subject of this paper.

References

1. Aquilano, C., Barbuti, R., Bocchetti, P., and Martelli, M. [1986] Negation as Failure: Completeness of the Query Evaluation Process for Horn Clause Programs with Recursive Definitions, *Journal of Automated Reasoning* **2**, 155–170.

2. *Artificial Intelligence* [1980] **13**(1).

3. Apt, K. R. and Emden, M. H. van [1982] Contributions to the Theory of Logic Programming, *JACM* **29**(3):841–862.

4. Apt, K. R. and Pugin, J. M. [1987] Maintenance of Stratified Databases Viewed as a Belief Revision System, in *Proc. of the 6th ACM Symposium on Principles of Database Systems*, San Diego, CA, 136–145.

5. Blair, H. A. [1982] Recursion Theoretic Complexity of the Semantics of Predicate Logic as a Programming Language, *Information and Control* **54**, 25–46.

6. Bidoit, N. and Hull, R. [1986] Positivism Versus Minimalism in Deductive Databases, *Proc. of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Cambridge, MA, 123–132.

7. Barbuti, R. and Martelli, M. [1986] Completeness of the SLDNF-Resolution for a Class of Logic Programs, *Proc. of the 3rd International Conference on Logic Programming*, Lecture Notes in Computer Science, No. 227, Springer-Verlag, Berlin, 600–613.

8. Brough, D. and Walker, A. [1984] Some Practical Properties of Logic Programming Interpreters, *Proc. of the Japan FGCS84 Conference*, 149–156.

9. Cavedon, L. and Lloyd, J. W. [1987] Completeness Results for SLDNF-Resolution, Tech. Rep. 87/9, Dept. of Computer Science, Melbourne University.

10. Clark, K. L. [1978] Negation as Failure, in *Logic and Databases* (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, 293–322.

11. Chandra, A. and Harel, D. [1985] Horn Clause Queries and Generalizations, *The Journal of Logic Programming* **2**(1):1–5.

12. Emden, M. H. van [1986] Quantitative Deduction and Its Fixpoint Theory, *Journal of Logic Programming* **3**(1):37–54.

13. Emden, M. H. van and Kowalski, R. A. [1976] The Semantics of Predicate Logic as a Programming Language, *JACM* **23**(4):733–742.

14. Fitting, M. [1985] Logic Programming Based on Logic, manuscript, Dept. of Math. and Computer Science, Lehman College, Bronx, NY.

15. Gabbay, D. [1985] Theoretical Foundations for Non-monotonic Reasoning in Expert Systems, in *Logics and Models of Concurrent Systems* (K. R. Apt, Ed.), Springer-Verlag, 439–458.

16. Gallier, J. and Raatz, S. [1988] A Graph-based Interpreter for General Horn Clauses, *Journal of Logic Programming*.

17. Hill, R. [1974] LUSH-Resolution and Its Completeness, *DCl Memo 78*, Department of Artificial Intelligence, University of Edinburgh.

18. Jaffar, J., Lassez, J.-L., and Lloyd, J. W. [1983] Completeness of the Negation as a Failure Rule, *IJCAI-83*, 500–506.

19. Kowalski, R. A. [1974] Predicate Logic as a Programming Language, *IFIP 74*, 569–574.

20. Kunen, K. [1987] Signed Data Dependencies in Logic Programs, Computer Sciences Technical Report 719, University of Wisconsin.

21. Lifschitz, V. [1988] On the Declarative Semantics of Logic Programs with Negation, in *Foundations of Deductive Databases and Logic Programming* (J. Minker, Ed.), Morgan Kaufmann, Publishers, Los Altos, CA, 177–192.

22. Lloyd, J. W. [1984] *Foundations of Logic Programming*, Springer-Verlag.

23. Lloyd, J. W., Sonenberg, E. A., and Topor, R. W. [1986] Integrity Constraint Checking in Stratified Databases, *Technical Report 86/5*, Dept. of Computer Science, University of Melbourne.

24. Lloyd, J. W. and Topor, R. W. [1985] A Basis for Deductive Database Systems II, *Journal of Logic Programming*, 3(1):55–67.

25. McCarthy, J. [1980] Circumscription—A Form of Nonmonotonic Reasoning, *Artificial Intelligence* **13**(1):295–323.

26. Minker, J. [1982] On Indefinite Databases and the Closed World Assumption, *Proc. of the 6th Conference on Automated Deduction* (D. W. Loveland, Ed.), Lecture Notes in Computer Science 138, Springer-Verlag, Berlin, 292–308.

27. Naqvi, S. A. [1986] A Logic for Negation in Database Systems, in *Proc. of the Workshop on Foundations of Deductive Databases and Logic Programming*, Washington, DC, 378–387

28. *Proc. of the AAAI Workshop on Non-monotonic Reasoning* [1984].

29. Przymusinski, T. [1988] On the Semantics of Stratified Deductive Databases, in *Foundations of Deductive Databases and Logic Programming* (J. Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 193–216.

30. Reiter, R. [1978] On Closed World Data Bases, in *Logic and Databases* (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, 55–76.

31. Rogers, H. Jr. [1967] *Theory of Recursive Functions and Effective Computability*, McGraw-Hill.

32. Sandewall, E. [1985] A Functional Approach to Non-monotonic Logic, *IJCAI-85*, 100–106.

33. Sebelik, J. and Stepanek, P. [1982] Horn Clause Programs for Recursive Functions, in *Logic Programming* (K. Clark and S.-A. Tärnlund, Eds.), Academic Press, 325–340.

34. Shepherdson, J. C. [1984] Negation as Failure: A Comparison of Clark's Completed Data Base and Reiter's Closed World Assumption, *Journal of Logic Programming* **1**(1):51-81.

35. Shepherdson, J. C. [1985] Negation as Failure II, *Journal of Logic Programming* **2**(2):185-202.

36. Shepherdson, J. C. [1988] Negation in Logic Programming, in *Foundations of Deductive Databases and Logic Programming* (J. Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 19-88.

37. Tärnlund, S. A. [1977] Horn Clause Compatibility, *BIT* **17**, 215-226.

38. Tarski, A. [1955] A Lattice-theoretical Fixpoint Theorem and Its Applications, *Pacific J. Math.* **5**, 285-309.

39. Topor, R. and Sonenberg, E. A. [1988] On Domain Independent Databases, in *Foundations of Deductive Databases and Logic Programming* (J. Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 217-240.

40. Van Gelder, A. [1988] Negation as Failure Using Tight Derivations for General Logic Programs, in *Foundations of Deductive Databases and Logic Programming* (J. Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 149-176.

41. Walker, A. [1986a] Syllog: An Approach to Prolog for Non-programmers, in *Logic Programming and Its Applications* (M. VanCaneghem and D.H.D. Warren, Eds.), Ablex, 32-49.

42. Walker, A. [1986b] A Knowledge Systems: Principles and Practice, *IBM Journal Res. Develop* **30**(1):2-13.