Stratified, Weak Stratified, and Three-valued Semantics*

Melvin Fitting
mlflc@cunyvm.cuny.edu
Dept. Mathematics and Computer Science
Lehman College (CUNY), Bronx, NY 10468
Depts. Computer Science, Philosophy, Mathematics
Graduate Center (CUNY), 33 West 42nd Street, NYC, NY 10036 †

Marion Ben-Jacob
Dept. of Math. and Comp. Inf. Science,
Mercy College, 555 Broadway,
Dobbs Ferry, NY 10522

Abstract

We investigate the relationship between three-valued Kripke/Kleene semantics and stratified semantics for stratifiable logic programs. We first show these are compatible, in the sense that if the three-valued semantics assigns a classical truth value, the stratified approach will assign the same value. Next, the familiar fixed point semantics for pure Horn clause programs gives both smallest and biggest fixed points fundamental roles. We show how to extend this idea to the family of stratifiable logic programs, producing a semantics we call *weak stratified*. Finally, we show weak stratified semantics coincides exactly with the three-valued approach on stratifiable programs, though the three-valued version is generally applicable, and does not require stratification assumptions.

1 Introduction

Logic programming using Horn clauses has a well-understood and generally accepted semantics based on classical logic ([20], [2]). But once negation is added to the machinery

*This is an expanded version of [10]

†Research partly supported by NSF Grant CCR-8702307 and PSC-CUNY Grant 6-67295.

things become more problematic. Two general fixed point approaches allowing negations have been introduced. One is that of *stratification*. In this, classical logic is still used, but certain restrictions are placed on allowed programs, so-called *stratifiability* restrictions ([1], [21], [3]). The other approach places no such restrictions, but substitutes a three-valued logic for the more familiar classical one ([17], [16], [4], [6], [14], [15]). Both approaches have their problems, but also both have natural motivations. In this paper we investigate the relationship between the two semantics, and we introduce a third version yet, which we call *weak stratification*.

We begin by introducing the machinery for the classical stratified, and the three-valued fixed point semantics, using the notion of work space, taken from [7]. This is a convenience, not a necessity, but we believe it provides natural and convenient machinery for logic programming investigations. We then prove that three-valued semantics is compatible with classical stratified semantics, on stratified programs. That is, if the three-valued version assigns a classical truth value (**t** or **f**, but not $\bot$), the stratified semantics must assign the same value.

In the classical fixed point semantics for programs without negation, [12] argues that the success set should be given by the least fixed point of what has become known as the T-operator, but the (ground) failure set by the *greatest* fixed point. This way of using the classical T-operator is essentially three-valued: not every ground atomic formula will be determined to succeed or to fail since there may be a gap between the least and the greatest fixed points. But in stratified semantics, the greatest fixed point plays no role. Loosely speaking, at each level of a stratification success is determined using the least fixed point of a T-operator associated with that portion of the program, but failure, or negation, is dealt with simply using complementation.

We define a modified version of stratified semantics that uses both least and greatest fixed points at every level of the stratification. The machinery needed for this is fairly elaborate. The T-operator uses classical logic, but when least and greatest fixed points are involved, the result can be three-valued, as observed above. This means that if we start off with a two-valued semantics at one level of stratification, we may have a three-valued semantics at the next, and this must somehow be converted back into a two-valued version so that a T-operator can be used once again. Nonetheless, there is a reasonable way of doing this, and a natural semantics emerges which we call the *weak stratified semantics*.

Our final result is that, for stratified programs, weak stratified semantics and three-valued semantics coincide.

There has been some understandable resistance to a semantics based on a three-valued logic. As Kunen says [15], "Many people will find a 3-valued logic not as natural or easy to understand as 2-valued logic. This is not a mathematical problem, but it does indicate a failure to give programmers a clear and understandable explanation of the declarative meaning of their Prolog programs." But one can argue that a three-valued logic was implicitly present in [2] all along, and was actually made explicit in [16]. And in fact, a three-valued logic is very natural to use when discussing the semantics of any programming language.

Think of the three values as true ($\mathbf{t}$), false ($\mathbf{f}$), and undefined ($\perp$). Suppose, in Pascal, we have an instruction that starts with "`if P and Q then ...`", where `P` and `Q` are functions that return Boolean. What will happen if `P` returns $\mathbf{f}$, but the function call `Q` never terminates? The easiest way to explain things is to say that many implementations of Pascal implicitly use a three-valued logic in which $\mathbf{f} \wedge \perp = \perp \wedge \mathbf{f} = \perp$. In fact, this logic is well-known: it is Kleene's weak three-valued logic [13]. Pure logic programming, without Prolog's particular control structure, can be thought of as using Kleene's strong three-valued logic [13] in which $\mathbf{f} \wedge \perp = \perp \wedge \mathbf{f} = \mathbf{f}$. LISP, on the other hand, uses a different three-valued logic, an asymmetric one in which $\mathbf{f} \wedge \perp = \mathbf{f}$, but $\perp \wedge \mathbf{f} = \perp$. The logic used by LISP falls between the weak and strong Kleene logics. By allowing ourselves to talk in terms of three-valued logics it becomes very simple to explain some elementary differences between programming languages.

We take the coincidence of weak stratified semantics and three-valued semantics, for stratified logic programs, as an argument for the innate naturalness of the three-valued approach. Also the three-valued approach applies to arbitrary programs, not just to stratified ones. And finally, the three-valued approach allows for generalizations to be considered, based on alternate three-valued logics (see, for instance, [6], [8] and [9]), a possibility that is difficult to explore if only classical machinery can be employed. Indeed in Section 7 we briefly consider a semantics based on the asymmetric logic of LISP. Use of this logic makes the theory no more complicated, but yields a semantics that is closer to real Prolog, with its particular search strategy.

2 Syntax

Logic programming is generally done over the Herbrand universe. But real Prologs also allow numbers as constants, and other domains such as words or infinite trees are possible (see [11]). For our purposes there is no need for restrictions to a Herbrand universe, or to any particular domain. Consequently we do things in considerable generality. Our presentation here follows [7].

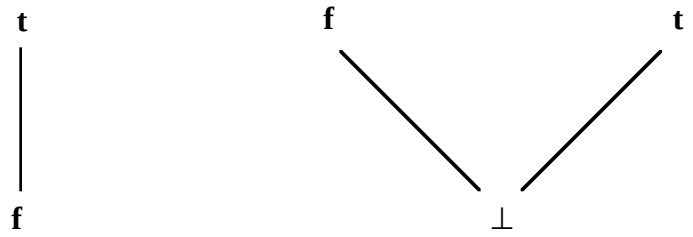
$\mathbf{D}$ is a non-empty domain, such as a Herbrand universe, or the positive integers. We need names for members of $\mathbf{D}$, to use in writing programs, so we use the logician's trick of allowing members of $\mathbf{D}$ in the formal language, to serve as names for themselves. With this understanding, we define a programming language $L(\mathbf{D})$.

We have an unlimited supply of *relation symbols*, of all arities, and an unlimited supply of *variables*. These are common to all languages $L(\mathbf{D})$.

A *term* of $L(\mathbf{D})$ is a variable or a member of $\mathbf{D}$. An *atomic formula* of $L(\mathbf{D})$ is an expression of the form $R(t_1, \ldots, t_n)$ where R is an n-place relation symbol and $t_1, \ldots, t_n$ are terms of $L(\mathbf{D})$. A *literal* of $L(\mathbf{D})$ is an atomic formula of $L(\mathbf{D})$ or the negation, $\neg A$, of an atomic formula A of $L(\mathbf{D})$.

A *program clause* of $L(\mathbf{D})$ is an expression $A \leftarrow B_1, \ldots, B_n$ where A is an atomic formula of $L(\mathbf{D})$, called the *head* and $B_1, \ldots, B_n$ is a list, possibly empty, of literals of $L(\mathbf{D})$, called

Figure 1: The partial orderings $\mathcal{TWO}$ and $\mathcal{THREE}$



the *body*. If $B_1,\ldots,B_n$ are all atomic, $A \leftarrow B_1, \ldots, B_n$ is a *Horn clause*. A *program* of $L(\mathbf{D})$ is a finite set of program clauses of $L(\mathbf{D})$. A *Horn clause program* of $L(\mathbf{D})$ is a finite set of Horn clauses of $L(\mathbf{D})$. Finally, a literal or a program clause is *ground* if it contains no variables.

3 Beginning semantics

Definition 3.1 $\mathcal{TWO}$ is the space of classical truth values $\{\mathbf{t}, \mathbf{f}\}$, with the ordering $<_2$ under which $\mathbf{f} <_2 \mathbf{t}$. $\mathcal{THREE}$ is the three-valued truth value space, $\{\mathbf{t}, \mathbf{f}, \bot\}$, with the ordering $<_3$ under which $\bot <_3 \mathbf{f}$ and $\bot <_3 \mathbf{t}$. ($\bot$ is read as undefined.) A *two-valued (or ordinary) relation* on a set $\mathbf{D}$ is a mapping $\mathbf{R}$ from $\mathbf{D}^n$ to $\mathcal{TWO}$. Likewise a *three-valued relation* is a mapping to $\mathcal{THREE}$.

Thus $<_2$ is an ordering based on degree of truth, while $<_3$ is one based on degree of information. These orderings are given schematically in Figure 1. This contrast between truth and information content receives a fuller treatment in [9].

The notion of work space for logic programs was introduced in [7] as a pedegogical device and to simplify the task of understanding program behavior. It is implicit in the notion of stratification.

Definition 3.2 A two-valued *work space* consists of:

1. a tuple, $\langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n \rangle$ where $\mathbf{D}$ is the *domain* and $\mathbf{R}_1, \ldots, \mathbf{R}_n$ are two-valued relations on $\mathbf{D}$, called *given* relations;
2. a pairing of a distinct relation symbol R_i with each given relation $\mathbf{R}_i$. These relation symbols are said to be *reserved* in the work space.

A three-valued work space is defined in the same way, but using three-valued relations.

The given relations of a three-valued work space can be thought of as *partial* relations, sometimes true, sometimes false, sometimes with an unknown truth value. Every two-valued

work space is trivially a three-valued one. We will denote a given relation by a bold face letter, like $\mathbf{R}$, and the relation symbol paired with it by a slant roman version, R. Then we need only specify a work space $\langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n \rangle$, leaving the reserved relation symbols to be understood.

Example 3.3 The following are examples of two-valued work spaces.

1. Domain: the set of non-negative integers. Given relation: $\mathbf{S}$, where $\mathbf{S}(x, y)$ iff $y = x+1$.
2. Domain: the Herbrand universe built up from a finite set of constant symbols using the one place function symbol f and the two-place function symbol g. Given relations: $\mathbf{F}$ and $\mathbf{G}$ where $\mathbf{F}(x, y)$ iff $y = f(x)$ and $\mathbf{G}(x, y, z)$ iff $z = g(x, y)$. This is a typical setting for conventional logic programming.

For interesting examples of three-valued work spaces, see [5]. In a work space we are *given* certain relations; we do not compute them. The following definition reflects this idea. In fact, the restriction has considerable technical import, which we can not go into here; see [7].

Definition 3.4 A program P of $L(\mathbf{D})$ is *in a work space* $\langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n \rangle$ if no reserved relation symbol appears in the head of any clause of P.

For many examples of such programs, see [7], Chapters 1 and 2. At a minimum a semantics for a logic program must supply an assignment of truth values to ground atomic formulas.

Definition 3.5 Suppose $\langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n \rangle$ is a two-valued work space. A *two-valued interpretation* is a mapping v from ground atomic formulas of $L(\mathbf{D})$ to $\mathcal{TWO}$. v is *in* the work space if, for each given relation $\mathbf{R}_i$, $v(R_i(\mathbf{a})) = \mathbf{R}_i(\mathbf{a})$. *Three-valued interpretations* are defined similarly, using $\mathcal{THREE}$.

Interpretations are given the 'pointwise' ordering. That is, for two-valued interpretations we take $v <_2 w$ provided $v(A) <_2 w(A)$ for each ground atomic formula A. Similarly for three-valued interpretations. Since $\mathcal{TWO}$ is a complete lattice, this makes the collection of two-valued interpretations into a complete lattice also. Then by the Knaster-Tarski Theorem [19], every order preserving map has both a smallest and a biggest fixed point. $\mathcal{THREE}$ is not a complete lattice. It is, however, a complete partial ordering, and more strongly a complete semi-lattice. This structure carries over to the collection of three-valued interpretations. By a generalization of the Knaster-Tarski Theorem, order preserving maps must have smallest, though not biggest, fixed points [4].

Definition 3.6 We use the notation μf to denote the least fixed point of f (if it exists) and νf to denote the greatest fixed point of f (again, provided it exists).

We want to introduce the standard two-valued fixpoint semantics, and also the three-valued version. We first need a few auxiliary definitions.

Definition 3.7 If P is a program of $L(\mathbf{D})$, by $P(\mathbf{D})$ we mean the set of all ground clauses that are substitution instances over $\mathbf{D}$ of program clauses of P. (In general, $P(\mathbf{D})$ will be infinite.)

Next, we say how to calculate the value of the body of a ground clause, under an interpretation. (The body of a ground clause is a list of literals.) We give the definition only for three-valued interpretations; this includes the two-valued version as a special case; the value $\perp$ is never taken on.

Definition 3.8 Suppose v is a three-valued interpretation in the work space $\langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n \rangle$.

Literal If $\neg A$ is a literal,

$$v(\neg A) = \begin{cases} \mathbf{t} & \text{if } v(A) = \mathbf{f} \\ \mathbf{f} & \text{if } v(A) = \mathbf{t} \\ \perp & \text{otherwise} \end{cases}$$

Empty List

$$v([\,]) = \mathbf{t}$$

Non-Empty List

$$v([A_1, \ldots, A_n]) = \begin{cases} \mathbf{t} & \text{if for every } i,\ v(A_i) = \mathbf{t} \\ \mathbf{f} & \text{if for some } i,\ v(A_i) = \mathbf{f} \\ \perp & \text{otherwise} \end{cases}$$

In [20] and [2] an operator T_P was associated with each Horn clause program P, mapping two-valued interpretations to two-valued interpretations. Loosely speaking, if v represents a state of knowledge, $T_P(v)$ is the state that results when the clauses in P are used once, starting from v. We make this more precise.

Definition 3.9 Let P be a Horn clause program in the two-valued work space $\langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n \rangle$. An operator T_P mapping two-valued interpretations to two-valued interpretations is defined as follows. If v is a two-valued interpretation:

Reserved Case

$$T_P(v)(R_i(\mathbf{a})) = \mathbf{R}_i(\mathbf{a})$$

Non-Reserved Case

$$T_P(v)(A) = \begin{cases} \mathbf{t} & \text{if there is a clause } A \leftarrow B \text{ in } P(\mathbf{D}), \text{ such that } v(B) = \mathbf{t} \\ \mathbf{f} & \text{if for every clause } A \leftarrow B \text{ in } P(\mathbf{D}),\ v(B) = \mathbf{f} \end{cases}$$

T_P maps two-valued interpretations *in* a work space to two-valued interpretations that are also in the work space, and T_P is order preserving using the ordering $\leq_2$, given that P is a Horn clause program. (The absence of negation is essential here.) Consequently T_P has both a smallest and a biggest fixed point. [2] shows that the smallest fixed point, μT_P, supplies us with the 'success set' for program P, while [12] shows the largest fixed point, νT_P, gives us the 'ground failure set'. In [4] we defined an analogous Φ_P operator on three-valued interpretations.

Definition 3.10 Let P be a program (allowing negations) in the three-valued work space $\langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n \rangle$. We associate with it a function Φ_P mapping three-valued interpretations to three-valued interpretations as follows. If v is a three-valued interpretation:

Reserved Case

$$\Phi_P(v)(R_i(\mathbf{a})) = \mathbf{R}_i(\mathbf{a})$$

Non-Reserved Case

$$\Phi_P(v)(A) = \begin{cases} \mathbf{t} & \text{if there is a clause } A \leftarrow B \text{ in } P(\mathbf{D}), \text{ such that } v(B) = \mathbf{t} \\ \mathbf{f} & \text{if for every clause } A \leftarrow B \text{ in } P(\mathbf{D}),\ v(B) = \mathbf{f} \\ \bot & \text{otherwise} \end{cases}$$

The operator T_P is classical and behaves as if atomic formulas in clause bodies were joined using the classical $\wedge$ operation, while multiple clauses having the same head act like their bodies were joined using the classical $\vee$. Similarly the operator Φ_P is based on Kleene's strong three-valued logic [13]. The negation operation used is Kleene's, and clause bodies, and multiple clauses are joined using Kleene's $\wedge$ and $\vee$ operations. See [4] for more details.

Φ_P is an order preserving operator on the space of three-valued interpretations, under the ordering $\leq_3$, so it has a least fixed point, $\mu\Phi_P$, though generally not a greatest one. In [4] we argued that this least fixed point gives a meaningful semantics for logic programs allowing negations. In particular we showed there was a strong connection between the two and three-valued semantics for programs not involving negations. We re-state that result, generalized to arbitrary (two-valued) work spaces.

Proposition 3.11 *Let P be a Horn clause program, in a two-valued workspace $\mathbf{W}$, and let A be a ground atomic formula.*

1. $(\mu\Phi_P)(A) = \mathbf{t}$ *iff* $(\mu T_P)(A) = \mathbf{t}$.

2. $(\mu\Phi_P)(A) = \mathbf{f}$ *iff* $(\nu T_P)(A) = \mathbf{f}$.

The smallest fixed point of a monotone operator in a complete semi-lattice, and also the smallest and the biggest fixed points in a complete lattice can be 'approximated to' via a transfinite sequence of steps. For the two-valued T_P operator, notation from [2] has become standard. We give a definition for it, and a related one for the three-valued Φ_P operator. This provides a tool for establishing Proposition 3.11, though in fact, Proposition 3.11 is generalized in Proposition 5.3, and we say more about its proof at that point.

Definition 3.12 Let P be a program in the three-valued work space $\mathbf{W}$. For each ordinal α we define a three-valued interpretation Φ_P^α as follows.

1. Φ_P^0 is the smallest three-valued interpretation in $\mathbf{W}$. That is, it assigns truth values to ground atomic formulas involving reserved relation symbols in accordance with the given relations of $\mathbf{W}$, and on unreserved relation symbols it is identically $\perp$;
2. $\Phi_P^{\alpha+1} = \Phi_P(\Phi_P^\alpha)$.
3. for a limit ordinal λ, $\Phi_P^\lambda = \sup\{\Phi_P^\alpha \mid \alpha < \lambda\}$.

Definition 3.13 Let P be a Horn clause program in the two-valued work space $\mathbf{W}$. For each ordinal α two-valued interpretations $T_P \uparrow^\alpha$ and $T_P \downarrow^\alpha$ are defined as follows.

1. $T_P \uparrow^0$ is the smallest two-valued interpretation in $\mathbf{W}$. It makes reserved relation symbols and given relations agree, and otherwise is identically $\mathbf{f}$;
2. $T_P \uparrow^{\alpha+1} = T_P(T_P \uparrow^\alpha)$;
3. for a limit ordinal λ, $T_P \uparrow^\lambda = \sup\{T_P \uparrow^\alpha \mid \alpha < \lambda\}$.

1. $T_P \downarrow^0$ is the biggest two-valued interpretation in $\mathbf{W}$. It makes reserved relation symbols and given relations agree, and otherwise is identically $\mathbf{t}$;
2. $T_P \downarrow^{\alpha+1} = T_P(T_P \downarrow^\alpha)$;
3. for a limit ordinal λ, $T_P \downarrow^\lambda = \inf\{T_P \downarrow^\alpha \mid \alpha < \lambda\}$.

The sequence Φ_P^α increases in the ordering $\leq_3$, to the least fixed point of Φ_P. Similarly $T_P \uparrow^\alpha$ increases in the ordering $\leq_2$ to the least fixed point of T_P, while $T_P \downarrow^\alpha$ decreases, and converges to the greatest fixed point of T_P.

4 Stratification

In [1] and [21], and earlier in [3], *stratification* was introduced into logic programming, based on the idea that relations must be completely defined before they can be used negatively. Not all programs are stratifiable, and for those that are the stratification need not be unique. Associated with the syntactical notion of stratification is a fixed point semantics, the so-called *stratified semantics*. It assigns a two-valued meaning to a stratified program that is independent of its stratification. In this section we state the syntactic stratification condition, and give a characterization of the classical stratified semantics in terms of work spaces. We also say a little about the effect of stratification on three-valued semantics.

Definition 4.1 A logic program P (allowing negations) has a *stratification* if the clauses of P can be divided into strata (in effect, subprograms) $P_1, \ldots, P_k$, so that:

1. each clause of P occurs in exactly one P_i;
2. all defining clauses for a relation symbol S occur in the same P_i;
3. if a clause is in P_i and if the unreserved relation symbol S occurs unnegated in the clause body, the defining clauses for S occur in P_j for some $j \leq i$;
4. if a clause is in P_i and if the unreserved relation symbol S occurs negated in the clause body, the defining clauses for S occur in P_j for some $j < i$.

A logic program is *stratifiable* if it has a stratification.

Stratifiability is a syntactic condition. We now present the corresponding classical semantics.

Definition 4.2 A program P in a work space $\mathbf{W}$ is *admissible* in $\mathbf{W}$ if all negations in clause bodies are applied to reserved relation symbols.

Admissibility is the simplest case of stratifiability, since negation is only used on given relations, never on those defined by the program, and so only one strata is needed. Now, if $\mathbf{W} = \langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n \rangle$ is a two-valued work space, we can expand it to represent negative information by adding the complement of each given relation as a new given relation. If $\mathbf{W} = \langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n \rangle$, we will denote by $\mathbf{W}*$ the work space $\langle \mathbf{D}, \mathbf{R}_1, \neg\mathbf{R}_1, \ldots, \mathbf{R}_n, \neg\mathbf{R}_n \rangle$, where $\neg\mathbf{R}_i$ is the complement of $\mathbf{R}_i$. Then, if the program P is admissible in $\mathbf{W}$, we can interpret it as an ordinary Horn clause program in $\mathbf{W}*$ by thinking of $\neg R_i$ as if it were a new relation symbol, and associating it with the relation $\neg\mathbf{R}_i$.

Definition 4.3 Suppose program P is admissible in the two-valued work space $\mathbf{W}$. Say the unreserved relation symbols of P are $S_1,\ldots,S_k$. By the *two-valued P-extension* of $\mathbf{W}$ we mean the two-valued work space that is like $\mathbf{W}$, but with additional given relations $\mathbf{S}_1,\ldots,\mathbf{S}_k$, where $\mathbf{S}_i(\mathbf{a}) = (\mu T_P)(S_i(\mathbf{a}))$, where in turn the least fixed point of T_P is evaluated in $\mathbf{W}*$.

Now, suppose P is a stratified program, in the two-valued work space $\mathbf{W}$, with stratification $P_1, \ldots, P_k$. We define a sequence of work spaces, $\mathbf{W}_0$, $\mathbf{W}_1$,...,$\mathbf{W}_k$ where: $\mathbf{W}_0 = \mathbf{W}$, and otherwise $\mathbf{W}_i$ is the two-valued P_i-extension of $\mathbf{W}_{i-1}$. (This makes sense since, it is easy to see, P_i will be admissible in $\mathbf{W}_{i-1}$.) The classical *stratified semantics* for the program P is that given by the final work space, $\mathbf{W}_k$. It should be noted that the definition we gave makes essential use of the particular stratification for P, but [1] proves the resulting semantics is independent of stratification.

The notion of stratification is syntactic, so we can talk about a stratified program, but consider a three-valued semantics. In such a case stratification is not necessary for supplying a semantical meaning; in three-valued semantics all logic programs can be handled without restriction. Still, the existence of a stratification does give us additional information.

Definition 4.4 Suppose program P is admissible in the three-valued work space $\mathbf{W}$. Say the unreserved relation symbols of P are S_1,...,S_k. By the *three-valued P-extension* of $\mathbf{W}$ we mean the three-valued work space that is like $\mathbf{W}$, but with additional given three-valued relations $\mathbf{S}_1$,...,$\mathbf{S}_k$, where $\mathbf{S}_i(\mathbf{a}) = (\mu\Phi_P)(S_i(\mathbf{a}))$, and where the least fixed point of Φ_P is evaluated in $\mathbf{W}$.

Definition 4.5 If v is a three-valued interpretation and $\mathcal{P}$ is a set of relation symbols, by $v\backslash\mathcal{P}$ (read v *restricted to* $\mathcal{P}$) we mean the interpretation that agrees with v on members of $\mathcal{P}$, and is $\bot$ otherwise.

What stratification tells us in the three-valued setting is that in computing meanings of relation symbols we won't need to consider anything from a higher strata. We make this more precise.

Proposition 4.6 *Let P be a stratified program in the three-valued work space $\mathbf{W}_0$, with P_1,..., P_k as a stratification. Let $\mathbf{W}_0$, $\mathbf{W}_1$,..., $\mathbf{W}_k$ be the sequence of three-valued work spaces such that $\mathbf{W}_i$ is the three-valued P_i-extension of $\mathbf{W}_{i-1}$. Let $\mathcal{P}_i$ be the set of reserved relation symbols together with those whose defining clauses are at level i or less in the stratification. Then, for all i, $\mu\Phi_{P_i} = \mu\Phi_P\backslash\mathcal{P}_i$, where $\mu\Phi_{P_i}$ is calculated in $\mathbf{W}_{i-1}$ but $\mu\Phi_P$ is calculated in $\mathbf{W}_0$.*

Proof The argument is by induction on i. We assume $\mu\Phi_{P_i} = \mu\Phi_P\backslash\mathcal{P}_i$ for $i < n$, and we show $\mu\Phi_{P_n} = \mu\Phi_P\backslash\mathcal{P}_n$. We do this in two halves by showing each side is $\leq_3$ the other. We sketch one half and leave the other to the reader. We show $\mu\Phi_{P_n} \leq_3 \mu\Phi_P\backslash\mathcal{P}_n$. And this can be done by a transfinite induction, showing for each ordinal α, $\Phi^\alpha_{P_n} \leq_3 \mu\Phi_P\backslash\mathcal{P}_n$.

If $\alpha = 0$, we have the following argument. $\Phi^0_{P_n} = \mu\Phi_{P_{n-1}}$ because $\Phi^0_{P_n}$ agrees with the given relations of $\mathbf{W}_{n-1}$ and is otherwise $\bot$, and these given relations are the ones determined by $\mu\Phi_{P_{n-1}}$. Next, $\mu\Phi_{P_{n-1}} \leq_3 \mu\Phi_P\backslash\mathcal{P}_{n-1}$ by the induction hypothesis, and this in turn is trivially $\leq_3 \mu\Phi_P\backslash\mathcal{P}_n$ since $\mathcal{P}_{n-1} \subseteq \mathcal{P}_n$. (We have omitted the straightforward $n = 1$ case.)

Suppose the result is known for α; we show it for $\alpha+1$. To do this we show that for each ground clause A, $\Phi_{P_n}^{\alpha+1}(A) \leq_3 (\mu\Phi_P \backslash \mathcal{P}_n)(A)$. If the left hand side is $\perp$, or if A involves a reserved relation symbol in $\mathbf{W}_{n-1}$ the result is trivial. Now suppose this is not the case, and say $\Phi_{P_n}^{\alpha+1}(A) = \mathbf{t}$. (The case of it being $\mathbf{f}$ is similar.) Then there is an instance of a clause in P_n, $A \leftarrow C$ with $\Phi_{P_n}^{\alpha}(C) = \mathbf{t}$. Using the transfinite induction hypothesis, $(\mu\Phi_P \backslash \mathcal{P}_n)(C) = \mathbf{t}$, and hence A is true under $\Phi_P(\mu\Phi_P \backslash \mathcal{P}_n)$. But $\Phi_P(\mu\Phi_P \backslash \mathcal{P}_n) \leq_3 \Phi_P(\mu\Phi_P) = \mu\Phi_P$, and so $\mu\Phi_P(A) = \mathbf{t}$. It then follows that $(\mu\Phi_P \backslash \mathcal{P}_n)(A) = \mathbf{t}$.

Finally, the limit ordinal case is straightforward, and is omitted.

We now show that the three-valued semantics is compatible with the classical stratified semantics, for stratified programs. That is, for a stratified program P, if the three-valued semantics assigns a classical truth value, the classical stratified semantics must assign the same value. The converse is not the case though, since the classical stratified semantics must always assign either $\mathbf{t}$ or $\mathbf{f}$, while the three-valued version can take on the value $\perp$. The following gives a precise statement of the result.

Proposition 4.7 *Suppose P is a stratified program in the two-valued work space $\mathbf{W}$. Let v be the meaning assigned to P by the classical stratified semantics. (v is a two-valued interpretation.) Then $\mu\Phi_P \leq_3 v$.*

Proof We begin with the following notation. For three-valued work spaces $\mathbf{A}$ and $\mathbf{B}$, we will write $\mathbf{A} \leq_3 \mathbf{B}$ provided: 1) $\mathbf{A} = \langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n \rangle$ and $\mathbf{B} = \langle \mathbf{D}, \mathbf{S}_1, \ldots, \mathbf{S}_n \rangle$ (thus they have the same domains), 2) for each i, $\mathbf{R}_i$ and $\mathbf{S}_i$ have the same arity (thus the two work spaces have the same signatures), 3) for each i and $\mathbf{a}$, $\mathbf{R}_i(\mathbf{a}) \leq_3 \mathbf{S}_i(\mathbf{a})$.

Now, suppose $P_1, \ldots P_k$ is a stratification of P. For the classical stratified semantics we construct a sequence of two-valued work spaces $\mathbf{W}_0$, $\mathbf{W}_1, \ldots$, $\mathbf{W}_k$, where $\mathbf{W}_0 = \mathbf{W}$ and $\mathbf{W}_i$ is the two-valued P_i extension of $\mathbf{W}_{i-1}$. We also can produce a sequence of three-valued work spaces $\mathbf{V}_0$, $\mathbf{V}_1, \ldots \mathbf{V}_k$ where $\mathbf{V}_0 = \mathbf{W}$ and $\mathbf{V}_i$ is the three-valued P_i extension of $\mathbf{V}_{i-1}$. We will show by induction on i that $\mathbf{V}_i \leq_3 \mathbf{W}_i$, and hence that $\mathbf{V}_k \leq_3 \mathbf{W}_k$. The classical stratified semantics for P is supplied by $\mathbf{W}_k$, while the three-valued model $\mathbf{V}_k$ gives us the three-valued semantics for P, via an application of Proposition 4.6.

We need a few results, of some interest in their own rights, that will make the proof quite simple.

Suppose $\mathbf{A}$ and $\mathbf{B}$ are three-valued work spaces, $\mathbf{A} \leq_3 \mathbf{B}$, and Q is any program. Then $\mu\Phi_Q$ evaluated in $\mathbf{A}$ $\leq_3$ $\mu\Phi_Q$ evaluated in $\mathbf{B}$. This can be shown by a straightforward transfinite induction, using the approximation sequences to $\mu\Phi_Q$.

If $\mathbf{A}$ is a two-valued work space, and Q is any program admissible in $\mathbf{A}$, then $\mu\Phi_Q$ evaluated in $\mathbf{A} \leq_3 \mu T_Q$ evaluated in $\mathbf{A}*$. This also can be proved using transfinite induction.

Now, the proof of the theorem is easy. $\mathbf{W}_0 = \mathbf{V}_0$ since both are $\mathbf{W}$. For the induction step, suppose $\mathbf{V}_{i-1} \leq_3 \mathbf{W}_{i-1}$. $\mathbf{V}_i$ is determined by the values assigned by $\mu\Phi_{P_i}$ evaluated in $\mathbf{V}_{i-1}$, and $\mathbf{W}_i$ is determined similarly by μT_{P_i} evaluated in $\mathbf{W}_{i-1}*$. But, $\mu\Phi_{P_i}$ evaluated in $\mathbf{V}_{i-1} \leq_3 \mu\Phi_{P_i}$ evaluated in $\mathbf{W}_{i-1}$, using the induction hypothesis and a result above, and this in turn is $\leq_3 \mu T_{P_i}$ evaluated in $\mathbf{W}_{i-1}*$.

5 Lower and upper work spaces

Three-valued semantics is compatible with the classical stratified semantics, but does not coincide with it. On the other hand, as we observed earlier, classical stratified semantics makes no use of the greatest fixed point of the operator T_P, and thus has no role for the ground failure set of [2]. We intend to find a place for this, and finally to exactly equate a semantics based on the classical T_P operator with the one provided by the three-valued Φ_P operator, for stratified programs. There is one problem we face at the very start, however. Generally there is a gap between the least and the greatest fixed points of T_P, which will have the effect of leaving certain formulas without truth values. This means that, even if we start with a two-valued work space, and use the classical T_P operator, at the next stage in the process of determining stratified semantics we may be faced with a three-valued work space. What we need is a method of associating two-valued work spaces with three-valued ones, so that we can restore a classical setting for the next stage of the process.

Suppose we have a partial relation $\mathbf{R}$ and we want to associate with it some classical relation, in order to get a classical work space. Suppose also that we have a program P that uses $\mathbf{R}$ as a given relation. If we want to determine the least fixed point of T_P, to establish what *must* be true, we want to minimize truth, taking something to be true only if we are forced to do so. In such a case we should take $R(\mathbf{a})$ to be true only if $\mathbf{R}$ says so. But on the other hand suppose we want the greatest fixed point of T_P, to establish what must be false. Now we want to minimize falsehood, and maximize truth. So we should take $R(\mathbf{a})$ to be true if $\mathbf{R}$ does not say otherwise, that is, if $\mathbf{R}(\mathbf{a})$ is not false. This suggests we need two work spaces, a lower one in which R is interpreted strictly, and an upper one in which R is interpreted as liberally as possible. Similar comments apply to $\neg R$.

Definition 5.1 Let $\mathbf{R}$ be a three-valued relation. We define four associated two-valued relations as follows. In the first two cases we give the condition for mapping to $\mathbf{t}$, which is enough to completely specify a two-valued relation. In the second two cases the negation is classical.

1. $\mathbf{R}_+(\mathbf{a}) = \mathbf{t} \Longleftrightarrow \mathbf{R}(\mathbf{a}) = \mathbf{t}$,
2. $\mathbf{R}_-(\mathbf{a}) = \mathbf{t} \Longleftrightarrow \mathbf{R}(\mathbf{a}) = \mathbf{f}$,
3. $\mathbf{R}^+(\mathbf{a}) = \neg\mathbf{R}_-(\mathbf{a})$,
4. $\mathbf{R}^-(\mathbf{a}) = \neg\mathbf{R}_+(\mathbf{a})$.

$\mathbf{R}_+$ and $\mathbf{R}_-$ represent the positive and negative information contained in $\mathbf{R}$. But $\mathbf{R}^+$ and $\mathbf{R}^-$ also do in a weak, default way. For example, $\mathbf{R}_+(\mathbf{a})$ is $\mathbf{t}$ exactly when $\mathbf{R}$ says it is, but $\mathbf{R}^+(\mathbf{a})$ is $\mathbf{t}$ just when $\mathbf{R}$ does not force it to be $\mathbf{f}$ (it is either $\mathbf{t}$ or $\perp$ as far as $\mathbf{R}$ is concerned). Similarly for $\mathbf{R}_-$, $\mathbf{R}^-$ and $\neg\mathbf{R}$. It is easy to check that $\mathbf{R}_+ \leq_2 \mathbf{R}^+$ and $\mathbf{R}_- \leq_2 \mathbf{R}^-$. If we think of $\mathbf{R}$ as partial information about a relation that is actually two-valued, $\mathbf{R}_+$ and $\mathbf{R}^+$ provide lower and upper bounds on when $\mathbf{R}$ holds; similarly $\mathbf{R}_-$ and $\mathbf{R}^-$ provide bounds on

R failing. Finally, using the consensus operation $\otimes$ for the bilattice $\mathcal{FOUR}$, as defined in [9], we have $\mathbf{R} = \mathbf{R}_+ \otimes \mathbf{R}^+$ and $\neg\mathbf{R} = \mathbf{R}_- \otimes \mathbf{R}^-$, where this negation is that of Kleene's three-valued logic. (We will make no use of this observation about bilattices here.)

Definition 5.2 Let $\mathbf{W} = \langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n \rangle$ be a three-valued work space. We associate with it a *lower* and an *upper* two-valued work space as follows. $\mathbf{W}_* = \langle \mathbf{D}, (\mathbf{R}_1)_+, (\mathbf{R}_1)_-, \ldots, (\mathbf{R}_n)_+, (\mathbf{R}_n)_- \rangle$. $\mathbf{W}^* = \langle \mathbf{D}, (\mathbf{R}_1)^+, (\mathbf{R}_1)^-, \ldots, (\mathbf{R}_n)^+, (\mathbf{R}_n)^- \rangle$. We call $\mathbf{W}_*/\mathbf{W}^*$ a *lower/upper work space pair*, and say it is *associated with* the three-valued work space **W**.

It is straightforward to reconstruct the three-valued work space **W** from the lower/upper pair $\mathbf{W}_*/\mathbf{W}^*$, and we will assume if either **W** or $\mathbf{W}_*/\mathbf{W}^*$ is given then both are known.

If R_i is a reserved relation symbol, in earlier sections we interpreted both it and its negation in $\mathbf{W}*$. We continue the same ideas here. We will pair R_i with the given relation $(\mathbf{R}_i)_+$ in $\mathbf{W}_*$, and with $(\mathbf{R}_i)^+$ in $\mathbf{W}^*$. But also we will treat $\neg R_i$ as if it were also a reserved relation symbol: in $\mathbf{W}_*$, $\neg R_i$ will be associated with $(\mathbf{R}_i)_-$, and in $\mathbf{W}^*$ with $(R_i)^-$. Thus even though negation symbols are present, they are being treated as syntactic devices producing 'funny' names for relations. We will think of admissible programs involving negations as if they were Horn programs, since negated relation symbols are being treated positively.

An admissible (or indeed any) program has a meaning in the three-valued sense. But using the device mentioned above for interpreting negations in $\mathbf{W}_*$ and $\mathbf{W}^*$, it also has meaning in these work spaces, in the two-valued sense. Here is a generalization of Proposition 3.11.

Proposition 5.3 *Let P be admissible in* **W**. *For a ground atomic formula A,*

1. $(\mu\Phi_P)(A) = \mathbf{t}$ *iff* $(\mu T_P)(A) = \mathbf{t}$, *where $\mu\Phi_P$ is calculated in* **W** *and μT_P is calculated in* $\mathbf{W}_*$,
2. $(\mu\Phi_P)(A) = \mathbf{f}$ *iff* $(\nu T_P)(A) = \mathbf{f}$ *where $\mu\Phi_P$ is calculated in* **W** *and νT_P is calculated in* $\mathbf{W}^*$.

Proof Transfinite induction is, once again, the appropriate tool. To show 1), for instance, one shows that, for each ordinal α, $\Phi_P^\alpha(A) = T_P \uparrow^\alpha (A)$. This is straightforward, and the details are omitted. Part 2) is similar, but using $T_P \downarrow^\alpha$ instead.

6 Weak stratified semantics

Classical stratified semantics uses *least* fixed points, and interprets negation using complementation. But the semantics of [2], which does not deal with negation directly, makes use of the *greatest* fixed point of T_P to determine the ground failure set. We combine these notions to produce a new semantics, which we call *weak stratified semantics*, that only uses the two-valued operator T_P, as does the classical approach, but that makes use of both least and greatest fixed points.

Definition 6.1 We will call a program P *admissible* in a lower/upper pair $\mathbf{W}_*/\mathbf{W}^*$ if it is admissible in the associated three-valued work space $\mathbf{W}$.

Definition 6.2 Suppose program P is admissible in the lower/upper pair $\mathbf{W}_*/\mathbf{W}^*$. Say the unreserved relation symbols of P are $S_1,\ldots, S_k$. By the *lower/upper P-extension* of $\mathbf{W}_*/\mathbf{W}^*$ we mean the lower/upper pair $\mathbf{V}_*/\mathbf{V}^*$, where $\mathbf{V}_*$ is like $\mathbf{W}_*$ but with the additional given relations $(\mathbf{S}_1)_+$, $(\mathbf{S}_1)_-,\ldots$, $(\mathbf{S}_k)_+$, $(\mathbf{S}_k)_-$, and $\mathbf{V}^*$ is like $\mathbf{W}^*$ but with the additional given relations $(\mathbf{S}_1)^+$, $(\mathbf{S}_1)^-,\ldots$, $(\mathbf{S}_k)^+$, $(\mathbf{S}_k)^-$, where in turn:

1. $(\mathbf{S}_i)_+(\mathbf{a}) = \mathbf{t} \Longleftrightarrow \mu T_P(S_i(\mathbf{a})) = \mathbf{t}$ in $\mathbf{W}_*$;
2. $(\mathbf{S}_i)_-(\mathbf{a}) = \mathbf{t} \Longleftrightarrow \nu T_P(S_i(\mathbf{a})) = \mathbf{f}$ in $\mathbf{W}^*$;
3. $(\mathbf{S}_i)^+(\mathbf{a}) = \neg(\mathbf{S}_i)_-(\mathbf{a})$;
4. $(\mathbf{S}_i)^-(\mathbf{a}) = \neg(\mathbf{S}_i)_+(\mathbf{a})$.

The lower/upper pair $\mathbf{W}_*/\mathbf{W}^*$ corresponds to a three-valued work space $\mathbf{W}$, and we can consider P as a program in $\mathbf{W}$ as well, using the three-valued semantics. We have notions of P-extension for both lower/upper work space pairs and for three-valued work spaces. (Only the first notion makes use of admissibility; it plays no role in the three-valued version.) Proposition 5.3 directly gives us the following.

Proposition 6.3 *Suppose the lower/upper pair $\mathbf{W}_*/\mathbf{W}^*$ and the three-valued work space $\mathbf{W}$ are associated, and P is a program that is admissible in $\mathbf{W}$. Then the three-valued P-extension of $\mathbf{W}$ and the lower/upper P-extension of $\mathbf{W}_*/\mathbf{W}^*$ are also associated.*

Now, suppose we have a logic program P in a work space $\mathbf{W}_0 = \langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n\rangle$. $\mathbf{W}_0$ can be a two or a three-valued work space; it adds no complications to deal with partial relations from the start. And suppose further that we have a stratification $P_1, \ldots, P_k$ of P. We use the program P, and its stratification, to define a sequence of lower/upper pairs, $(\mathbf{W}_0)_*/(\mathbf{W}_0)^*$, $(\mathbf{W}_1)_*/(\mathbf{W}_1)^*$, $\ldots$, $(\mathbf{W}_k)_*/(\mathbf{W}_k)^*$ as follows.

1. $(\mathbf{W}_0)_*$ and $(\mathbf{W}_0)^*$ are the lower and upper work spaces associated with $\mathbf{W}_0$. P_1 will be admissible in $(\mathbf{W}_0)_*/(\mathbf{W}_0)^*$.

2. Suppose we have defined $(\mathbf{W}_{i-1})_*/(\mathbf{W}_{i-1})^*$, and P_i is admissible in $(\mathbf{W}_{i-1})_*/(\mathbf{W}_{i-1})^*$. Then we take $(\mathbf{W}_i)_*/(\mathbf{W}_i)^*$ to be the lower/upper P_i-extension of $(\mathbf{W}_{i-1})_*/(\mathbf{W}_{i-1})^*$. P_{i+1} will be admissible in $(\mathbf{W}_i)_*/(\mathbf{W}_i)^*$.

Thus we produce a sequence of lower/upper pairs, and we can take the final one $(\mathbf{W}_k)_*/(\mathbf{W}_k)^*$ as supplying a "meaning" for the program P itself. It is the meaning assigned by this last lower/upper pair that we refer to as the *weak stratified semantics*.

One disadvantage of the standard notion of stratified semantics is obvious. Negation is treated via complementation, but since the relations that can be represented using Horn

clause programs are exactly the recursively enumerable ones, complements are not generally computable. [2] gives a simple example of a program for which the corresponding T_P operator is not 'down continuous'. This example can be used to show that computability problems still arise for weak stratification, though not in such a straightforward way. In general, approximations to greatest fixed points that 'cut off' after ω steps are reasonable to consider, from a computational point of view. One advantage of weak stratification is that the machinery is present to consider such truncation issues. This is not the case with the standard stratified semantics, which reduces negations to complementations in one step, and leaves no role at all for a process of approximation.

By using lower/upper pairs, and the classical T_P operator, we have assigned a three-valued meaning to a stratified program P. The assignment requires the construction of a sequence of work space pairs, one for each level of stratification. On the other hand Φ_P also assigns a meaning to P, whether stratified or not, and does so in a single work space.

Proposition 6.4 *For a stratified program P in work space $\mathbf{W}$, the meaning assigned using the weak stratified semantics, and the meaning assigned using the three-valued semantics are the same.*

Proof Suppose P_1, P_2,..., P_k is a stratification of P. To determine the weak stratified semantics for P we construct a sequence of lower/upper work space pairs, $(\mathbf{W}_0)_*/(\mathbf{W}_0)^*$, $(\mathbf{W}_1)_*/(\mathbf{W}_1)^*$, ..., $(\mathbf{W}_k)_*/(\mathbf{W}_k)^*$, where $(\mathbf{W}_0)_*/(\mathbf{W}_0)^*$ is associated with $\mathbf{W}$, and otherwise $(\mathbf{W}_i)_*/(\mathbf{W}_i)^*$ is the lower/upper P_i-extension of $(\mathbf{W}_{i-1})_*/(\mathbf{W}_{i-1})^*$. We can also construct a sequence of three-valued work spaces, $\mathbf{W}_0$, $\mathbf{W}_1$,..., $\mathbf{W}_k$, where $\mathbf{W}_0 = \mathbf{W}$, and otherwise $\mathbf{W}_i$ is the three-valued P_i-extension of $\mathbf{W}_{i-1}$. Using Proposition 6.3, corresponding members of these two sequences will be associated, hence $(\mathbf{W}_k)_*/(\mathbf{W}_k)^*$ and $\mathbf{W}_k$ will be associated. But $(\mathbf{W}_k)_*/(\mathbf{W}_k)^*$ supplies the weak stratified meaning, and $\mathbf{W}_k$ supplies the three-valued meaning, via an application of Proposition 4.6.

Corollary 6.5 *The meaning assigned to a stratified program using two valued lower/upper pairs is independent of the stratification.*

7 Conclusions

We have presented a weak stratified semantics, based on two-valued classical logic, applicable to stratified logic programs. We have argued that it is a better candidate for a program semantics than the conventional stratified semantics, because its treatment of negation makes use of greatest fixed points rather than complements. We have also shown that our version of stratified semantics agrees fully with the three-valued semantics based on Kleene's logic. This is significant for several reasons. The three-valued approach is considerably simpler in terms of the machinery required. And it is more general, since it applies to all programs, not just to stratified ones. For example, although $p \leftarrow p$ is stratified, $p \leftarrow \neg p$ is not. Still

the three-valued semantics is applicable, and assigns p the value $\perp$ as might be expected. As another example, consider the program

$even(0) \leftarrow .$

$even(s(X)) \leftarrow \neg even(X).$

This is not stratified. It is *locally* stratified though [18], and it is easy to check that the meaning assigned via the three-valued approach coincides with that assigned via local stratification. It would be of interest to establish a general relationship between these semantical approaches. A basic point stands out: the three-valued semantics is generally applicable, and does not need modification or extension every time a wider class of programs is considered.

There are computability problems associated with several of the semantics. Three-valued semantics may lead to relations that are not recursively enumerable, and hence so may weak stratified semantics. This is an issue with conventional stratified semantics as well, essentially because the recursively enumerable relations are not closed under complementation. But for the three-valued version an attractive alternative is available. The natural cut-off point for computability is after ω steps in the approximation sequence. In other words, work with Φ_P^ω instead of the least fixed point of Φ_P, or equivalently work with $T_{P_i} \uparrow^\omega$ and $T_{P_i} \downarrow^\omega$. [14] and [15] give a completeness result based on a variation of this idea which makes it very attractive indeed.

Finally, the three-valued approach is susceptible to further generalization, because other three-valued logics are around. In [6] we considered a three-valued logic based on supervaluations, and showed it coincided with a proof procedure based on semantic tableaux allowing recursive calls. In [9] a large family of logic programming languages is investigated, using multiple-valued logics. Finally, another attractive possibility is to use the asymmetric three-valued logic that LISP uses, in which conjunctions and disjunctions are evaluated from left to right. The whole three-valued approach works using this asymmetric logic too, because the essential monotonicity conditions on $\wedge$ and $\vee$ are still met. And the resulting semantics is closer to real Prolog, because the truth conditions for $\wedge$ and $\vee$ amount to a left-right evaluation of clause bodies, and a first-to-last consideration of clauses. We hope to investigate this asymetric semantics further elsewhere.

References

[1] K. R. Apt, H. A. Blair, A. Walker, Towards a theory of declarative knowledge, *Foundations of Deductive Databases and Logic Programming*, J. Minker ed, Morgan Kaufmann, pp 89–148, Los Altos (1987).

[2] K. R. Apt, M. H. Van Emden, Contributions to the theory of logic programming, *J. Assoc. Comput. Mach.*, vol 29, pp 841–863 (1982).

[3] A. K. Chandra, D. Harel, Horn clause queries and generalizations, *J. Logic Programming*, vol 2, pp 1–15 (1985).

[4] M. C. Fitting, A Kripke-Kleene semantics for logic programs, *J. Logic Programming*, vol 3, pp 93–114 (1986).

[5] M. C. Fitting, Logic programming semantics using a compact data structure, *Proceedings of the ACM SIGART International Symposium on Methodologies for Intelligent Systems*, Z. W. Ras and M. Zemankova, pp 247–255 (1986).

[6] M. C. Fitting, Partial models and logic programming, *Theoretical Computer Science*, vol 48, pp 229–255 (1986).

[7] M. C. Fitting, *Computability Theory, Semantics, and Logic Programming*, Oxford University Press, New York (1987).

[8] M. C. Fitting, Logic programming on a topological bilattice, *Fundamenta Informatica*. vol 11, pp 209–218 (1988).

[9] M. C. Fitting, Bilattices and the semantics of logic programming, to appear in *Journal of Logic Programming*.

[10] M. C. Fitting, M. Ben-Jacob, Stratified and Three-valued Logic Programming Semantics, *Logic Programming, Proc. of the Fifth Intl. Conf. and Symp.*, editors R. A. Kowalski and K. A. Bowen, pp 1054–1069, The MIT Press (1988).

[11] J. Jaffar, J.-L. Lassez, Constraint Logic Programming, *POPL* (1987).

[12] J. Jaffar, J.-L. Lassez, M. J. Maher, A logic programming language scheme, *Logic Programming: Relations, Functions and Equations*, D. DeGroot and G. Lindstrom editors, Prentice-Hall (1986).

[13] S. C. Kleene, *Introduction to Metamathematics*, Van Nostrand, Princeton (1952).

[14] K. Kunen, Negation in logic programming, *J. Logic Programming*, vol 4, pp 289–308 (1987).

[15] K. Kunen, Signed data dependencies in logic programs, forthcoming in *J. Logic Programming*.

[16] J.-L. Lassez, M. J. Maher, Optimal fixedpoints of logic programs, *Theoretical Computer Science*, vol 39, pp 15–25 (1985), reprinted from *FST-TCS Conference*, Bangalore (1983).

[17] A. Mycroft, Logic programs and many-valued logics, *Proc. 1st STACS Conf.* (1983).

[18] T. Przymusinski, On the declarative semantics of deductive databases and logic programs, *Foundations of Deductive Databases and Logic Programming*, J. Minker ed, Morgan Kaufmann, pp 193–216, Los Altos (1987).

[19] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, *Pacific Journal of Mathematics*, vol 5, pp 285–309 (1955).

[20] M. H. Van Emden, R. A. Kowalski, The semantics of predicate logic as a programming language, *J. ACM*, vol 23, pp 733–742 (1976).

[21] A. Van Gelder, Negation as failure using tight derivations for general logic programs, *Proc. 3rd IEEE Symp. on Logic Programming*, Salt Lake City, pp 127–138 (1986).