Stratified and Three-valued Logic Programming Semantics

Melvin Fitting*
Marion Ben-Jacob†

Abstract

The familiar fixed point semantics for Horn clause programs gives both smallest and biggest fixed points fundamental roles. We show how to extend this idea to the family of stratified logic programs, producing a semantics we call *weak stratified*, that is compatible with but not the same as the conventional stratified semantics. And we show weak stratified semantics coincides with one based on three valued logic, a semantics that is generally applicable, and that does not require stratification assumptions.

1 Introduction

A beautiful fixed point semantics for Horn clause logic programming has been developed, based on classical logic ([16], [2]). But it can not deal adequately with negations when they are allowed in clause bodies. Two kinds of generalizations have been proposed to deal with this problem. The best known is *stratification* [1], [17]. Here the kind of logic programs one is allowed to write is restricted; recursions through negations are forbidden. For such programs there is a fixed point semantics generalizing pure Horn clause semantics, yielding a classical, two valued semantics. The other kind of generalization involves a three valued logic [4], [11], [12] and thus moves away from classical logic. This approach has received somewhat less attention. As Kunen says [12], "Many people will find a 3-valued logic not as natural or easy to

*Dept. of Math. and Comp. Science, Lehman College, Bedford Park Blvd. West, Bronx, NY 10468

†Dept. of Math. and Comp. Inf. Science, Mercy College, 555 Broadway, Dobbs Ferry, NY 10522

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."

As shown in [9], both least and greatest fixed points play a role in assigning meanings to Horn clause logic programs. The least fixed point gives the 'success set' while the greatest one gives the 'ground failure set'. Stratified semantics, on the other hand, finds no role for greatest fixed points, and treats negations using the simple device of complementation. Since the recursively enumerable sets are not closed under complementation, this leads to obvious computability problems.

We propose an alternative fixed point semantics that is applicable to stratified logic programs, which we call *weak stratified semantics*. It uses only classical two valued models, and the T_P operator from [16] and [2], but it gives greatest fixed points a fundamental role to play in the interpretation of negation, thus exploiting the insights of [2] and [9]. While the machinery required is more complex, the results are natural and flexible. Just as with the conventional version of stratified semantics, weak stratified semantics is independent of the particular stratification used for a given logic program. And it can be proved that the truth values assigned by weak stratified semantics are compatible with those assigned by the conventional version of stratified semantics.

Although weak stratified semantics uses machinery of classical two valued logic, the way it is used leaves some formulas without a truth value. Thus a three valued logic emerges — indeed it was implicitly present in [2] all along, and is made explicit in [13]. 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 Pascal uses a three valued logic in which $\mathbf{f} \wedge \perp = \perp \wedge \mathbf{f} = \perp$. In fact, the logic Pascal uses is well-known. It is Kleene's weak three valued logic [10]. Pure logic programming, without Prolog's particular control structure, can be thought of as using Kleene's strong three valued logic [10] 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.

In [14], [13], [4], [5], [11], [12] a semantics for logic programs with negations was developed, based on Kleene's strong three valued logic. This semantics, and weak stratified semantics agree completely. This is important for several reasons. The coincidence of different approaches argues for innate naturalness. There is a completeness result for three valued semantics in [12], which is of major significance. The three valued approach applies to arbitrary programs, not just to stratified ones. And the three valued approach allows for generalizations to be considered, based on alternate three valued logics (see, for instance, [5] and [7]), a possibility that is difficult to explore if only classical machinery can be employed. Indeed in Section 8 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 a Herbrand universe. But real Prologs also allow numbers as constants, and other domains such as words or infinite trees are possible (see [8]). 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 [6].

D is a non-empty domain, such as a Herbrand universe, or the positive integers. We need names for members of **D**, to use in writing programs, so we use the logician's trick of allowing members of **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 **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 $L(\mathbf{D})$ and $B_1, \ldots, B_n$ is a list, possibly empty, of literals of $L(\mathbf{D})$. 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 $L(\mathbf{D})$.

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, $\{\mathbf{t}, \mathbf{f}, \perp\}$, with the ordering $<_3$ under which $\perp <_3 \mathbf{f}$ and $\perp <_3 \mathbf{t}$. ($\perp$ 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. The notion of work space for logic programs was introduced in [6] 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 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 3.1 *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 setting for conventional logic programming.*

Definition 3.3 *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 [6], 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 3.4 *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 (variable-free) 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, 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].

In [16] 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 3.5 *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.)*

Definition 3.6 *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 on two valued interpretations is defined as follows. If v is a two valued interpretation, $T_P(v) = w$ where, for a reserved relation symbol R_i, $w(R_i(\mathbf{a})) = \mathbf{R}_i(\mathbf{a})$; and for a non-reserved relation symbol S, $w(S(\mathbf{a})) = \mathbf{t}$ iff there is a Horn clause in $P(\mathbf{D})$ whose head is $S(\mathbf{a})$, such that each atomic formula in the clause body is mapped to $\mathbf{t}$ by v.*

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. Consequently T_P has both a smallest and a biggest fixed point. [2] shows that the smallest fixed point of T_P supplies us with the 'success set' for program P, while [9] shows the largest fixed point gives us the 'ground failure set'. In [4] we defined an analogous Φ_P operator on three valued interpretations.

Definition 3.7 *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 mapping Φ_P on the space of three valued interpretations. Let $\Phi_P(v) = w$ where, for a reserved relation symbol R_i, $w(R_i(\mathbf{a})) = \mathbf{R}_i(\mathbf{a})$. For a non-reserved relation symbol S, $w(S(\mathbf{a})) = \mathbf{t}$ if there is a clause in $P(\mathbf{D})$ whose head is $S(\mathbf{a})$ and whose body is mapped to $\mathbf{t}$ by the input interpretation v; $w(S(\mathbf{a})) = \mathbf{f}$ if every clause in $P(\mathbf{D})$ whose head is $S(\mathbf{a})$ has a body that v maps to $\mathbf{f}$; and $w(S(\mathbf{a})) = \perp$ otherwise. The value that v assigns to a clause body is $\mathbf{t}$ provided for every unnegated literal A in the clause body, A maps to $\mathbf{t}$, and for every negated literal $\neg A$ in the clause body A maps to $\mathbf{f}$ under v. Likewise v maps a clause body to $\mathbf{f}$ provided for some unnegated literal A in the body A maps to $\mathbf{f}$, or for some negated literal $\neg A$ in the body A maps to $\mathbf{t}$.*

The operator Φ_P is based on Kleene's strong three valued logic, though this may not be immediately apparent because of the way we defined things above. A logic program that has many clauses for a particular relation symbol can be converted into one with a single clause for it, provided a more general notion of clause is allowed. A comma is treated as as an $\wedge$; multiple clauses are joined using $\vee$; 'extra' variables are thought of as existentially quantified. (These steps are familiar from Clark's completed data base.) In this way a program converts into one in which each relation symbol occurs in at most one head, but bodies can be more general formulas of logic. Now the operator Φ_P can be re-defined in the following simple way: $\Phi_P(v) = w$ where, for an unreserved relation symbol S that occurs in a head in P, say in $S(\mathbf{x}) \leftarrow B(\mathbf{x})$, $w(S(\mathbf{a}))$ has the same truth value that $B(\mathbf{a})$ has, when atomic formulas are given truth values according to v, and connectives and quantifiers are evaluated using Kleene's strong three-valued logic. See [4] for more details.

Φ_P is an order preserving operator on the space of three-valued interpretations, so it has a least fixed point, 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 3.1 *Let P be a Horn clause program, in a two valued workspace $\mathbf{W}$, and let A be a ground atomic formula.*

1. *A is true in the least fixed point of Φ_P iff A is true in the least fixed point of T_P.*
2. *A is false in the least fixed point of Φ_P iff A is false in the greatest fixed point of T_P.*

4 An elementary generalization

As a first step in extending the ideas of [2] to cover stratified programs we make a minimal generalization to allow three valued work spaces, and negations applied to reserved relation symbols. If we have a three valued work space we must associate some two valued work space with it so that the T_P operator can used. Suppose we have a partial relation $\mathbf{R}$ and a program P that uses it as a given relation. And suppose we want to determine the least fixed point of T_P, to establish what *must* be true. This means we want to minimize truth, taking something to be true only if we are forced to do so. Hence 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 4.1 *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. $\mathbf{R}_+(\mathbf{a}) = \mathbf{t} \iff \mathbf{R}(\mathbf{a}) = \mathbf{t}$,
2. $\mathbf{R}_-(\mathbf{a}) = \mathbf{t} \iff \mathbf{R}(\mathbf{a}) = \mathbf{f}$,
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 $\mathbf{t}$ if $\mathbf{R}$ does not force it to be $\mathbf{f}$ (it is either $\mathbf{t}$ or $\perp$ as far as $\mathbf{R}$ is concerned). Trivially $\mathbf{R}_+ \leq_2 \mathbf{R}^+$ 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 $\mathbf{R}$ failing.

Definition 4.2 *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* $\mathbf{W}$.

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

If R_i is a reserved relation symbol, we will pair it 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 programs involving negations as if they were Horn programs, since negated relation symbols are being treated positively.

Definition 4.3 *A program* P *in a three valued work space* $\mathbf{W}$ *is* admissible *in* $\mathbf{W}$ *if all negations in clause bodies are of reserved relation symbols.* P *is admissible in a lower/upper work space pair if it is admissible in the associated three valued work space.*

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 3.1.

Proposition 4.1 *Let* P *be admissible in* $\mathbf{W}$. *For a ground atomic formula* A,

1. A *maps to* $\mathbf{t}$ *using the least fixed point of* Φ_P *in* $\mathbf{W} \iff A$ *maps to* $\mathbf{t}$ *using the least fixed point of* T_P *in* $\mathbf{W}_*$,
2. A *maps to* $\mathbf{f}$ *using the least fixed point of* Φ_P *in* $\mathbf{W} \iff A$ *maps to* $\mathbf{f}$ *using the greatest fixed point of* T_P *in* $\mathbf{W}^*$.

The smallest fixed point of a monotone operator in a complete semilattice, 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 4.1.

Definition 4.4 *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 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 4.5 *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 to the least fixed point of Φ_P. Similarly for $T_P \uparrow^\alpha$, while $T_P \downarrow^\alpha$ is decreasing, and converges to the greatest fixed point of T_P. Then Proposition 4.1 is an immediate consequence of the following, which has a straightforward proof by transfinite induction.

Proposition 4.2 *Let P be a logic program that is admissible in the three valued workspace $\mathbf{W}$. Let A be a ground atomic formula. Then, for each ordinal α,*

1. *A maps to* $\mathbf{t}$ *under* Φ_P^α *in* $\mathbf{W} \iff A$ *maps to* $\mathbf{t}$ *under* $T_P \uparrow^\alpha$ *in* $\mathbf{W}_*$,
2. *A maps to* $\mathbf{f}$ *under* Φ_P^α *in* $\mathbf{W} \iff A$ *maps to* $\mathbf{f}$ *under* $T_P \downarrow^\alpha$ *in* $\mathbf{W}^*$.

5 Work space extensions

For this section suppose we have a program P that is admissible in the lower/upper work space pair $\mathbf{W}_*/\mathbf{W}^*$, where $\mathbf{W}_* = \langle \mathbf{D}, \mathbf{R}_{1+}, \mathbf{R}_{1-}, \ldots, \mathbf{R}_{n+}, \mathbf{R}_{n-} \rangle$ and $\mathbf{W}^* = \langle \mathbf{D}, \mathbf{R}_1^+, \mathbf{R}_1^-, \ldots, \mathbf{R}_n^+, \mathbf{R}_n^- \rangle$. Say the unreserved relation symbols of P are $S_1, \ldots, S_k$. As in Section 4 we can use the T_P operator to assign meanings to these relation symbols.

Definition 5.1 *For* $i = 1, \ldots, k$*:*

1. $\mathbf{S}_{i+}(\mathbf{a}) = \mathbf{t} \iff S_i(\mathbf{a})$ *maps to* $\mathbf{t}$ *using the least fixed point of* T_P *in* $\mathbf{W}_*$*;*
2. $\mathbf{S}_{i-}(\mathbf{a}) = \mathbf{t} \iff S_i(\mathbf{a})$ *maps to* $\mathbf{f}$ *using the greatest fixed point of* T_P *in* $\mathbf{W}^*$*;*
3. $\mathbf{S}_i^+(\mathbf{a}) = \neg \mathbf{S}_{i-}(\mathbf{a})$*;*
4. $\mathbf{S}_i^-(\mathbf{a}) = \neg \mathbf{S}_{i+}(\mathbf{a})$.

We define a new lower/upper pair as follows:

1. $\mathbf{V}_* = \langle \mathbf{D}, \mathbf{R}_{1+}, \mathbf{R}_{1-}, \ldots, \mathbf{R}_{n+}, \mathbf{R}_{n-}, \mathbf{S}_{1+}, \mathbf{S}_{1-}, \ldots, \mathbf{S}_{k+}, \mathbf{S}_{k-} \rangle$,
2. $\mathbf{V}^* = \langle \mathbf{D}, \mathbf{R}_1^+, \mathbf{R}_1^-, \ldots, \mathbf{R}_n^+, \mathbf{R}_n^-, \mathbf{S}_1^+, \mathbf{S}_1^-, \ldots, \mathbf{S}_k^+, \mathbf{S}_k^- \rangle$.

We refer to $\mathbf{V}_*/\mathbf{V}^*$ *as the* P-extension *of* $\mathbf{W}_*/\mathbf{W}^*$.

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.

Definition 5.2 *For* $i = 1, \ldots, k$, $\mathbf{S}_i(\mathbf{a})$ *is the truth value of* $S_i(\mathbf{a})$ *in the least fixed point of* Φ_P *in* $\mathbf{W}$*. We define a new three valued work space as follows.* $\mathbf{V} = \langle \mathbf{D}, \mathbf{R}_1, \ldots, \mathbf{R}_n, \mathbf{S}_1, \ldots, \mathbf{S}_k \rangle$*. We also refer to* $\mathbf{V}$ *as the* P-extension *of* $\mathbf{W}$.

Thus 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 4.1 immediately gives us the following.

Proposition 5.1 *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* P*-extension of* $\mathbf{W}$ *and the* P*-extension of* $\mathbf{W}_*/\mathbf{W}^*$ *are also associated.*

6 Weak stratified semantics

In [1] and [17], 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 present a somewhat more complex semantics for stratified programs which we call *weak stratified semantics* that gives both smallest and biggest fixed points roles to play. In the next section we show it too assigns meanings to stratified programs that are independent of the particular stratification. And we show the resulting semantics is equivalent to the three valued semantics for stratifiable programs.

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 of P. That is, 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 $j < i$.

Now 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 $\mathbf{W}_{i*}/\mathbf{W}_i^*$, and P_{i+1} is admissible in $\mathbf{W}_{i*}/\mathbf{W}_i^*$. Let $\mathbf{W}_{i+1*}/\mathbf{W}_{i+1}^*$ be the P_{i+1}-extension of $\mathbf{W}_{i*}/\mathbf{W}_i^*$. P_{i+2} will be admissible in $\mathbf{W}_{i+1*}/\mathbf{W}_{i+1}^*$.

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*.

Our use of lower/upper pairs has given both the smallest and the biggest fixed points of the T_P operator roles to play. In the standard stratified semantics only the smallest fixed point is used. We can get the same effect here via a simple modification. First, suppose the initial work space $\mathbf{W}_0$ is two valued. Then, carry out the construction as above, but with every occurrence of "greatest fixed point" replaced by an occurrence of "least fixed point". This makes lower and upper work spaces in the sequence identical, and so the machinery can be simplified. Now, the meaning assigned by the last term in the sequence is (equivalent to) the conventional notion of stratified semantics.

The disadvantage of the standard notion of stratified semantics is obvious. Negation is treated semantically 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 shows 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. The advantage of weak stratification is that the machinery is present to consider 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.

7 Connections

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 7.1 *For a stratified program P the meaning assigned using the weak stratified semantics, and the meaning assigned using Φ_P are the same.*

To prove this one needs Proposition 5.1 and the following, whose (omitted) proof is by induction on i, within which is a transfinite induction on the sequence of approximations to the least fixed points.

Lemma 7.2 *Let P be a stratified program in 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 P_i extension of $\mathbf{W}_{i-1}$. Suppose S is an unreserved relation symbol whose defining clauses are in P_i. Then the truth value of $S(\mathbf{a})$ in the least fixed point of Φ_{P_i}, calculated in $\mathbf{W}_{i-1}$ is the same as the truth value of $S(\mathbf{a})$ in the least fixed point of Φ_P, calculated in $\mathbf{W}_0$.*

Corollary 7.3 *The meaning assigned to a stratified program using a sequence of two valued lower/upper pairs is independent of the stratification.*

8 Conclusions

We have presented a 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. Further it can be shown, though we do not do so here, that whenever the weak stratified semantics assigns a truth value of $\mathbf{t}$ or $\mathbf{f}$, the conventional stratified semantics will assign the same truth value. (The converse is not true since the conventional stratified semantics does not use $\perp$ and so must assign $\mathbf{t}$ or $\mathbf{f}$ to everything.) Thus our version of stratified semantics is consistent with the conventional version, but narrower in its assignment of truth values. For instance, using the familiar program $p \leftarrow p$, the conventional semantics assigns to p the value $\mathbf{f}$, whereas ours leaves p without a truth value, $\perp$ in other words.

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 [15], 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.

We mentioned in Section 6 the problem of computability: weak stratified, or three-valued semantics may lead to relations that are not recursively enumerable. This is an issue with conventional stratified semantics as well. 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 $T_{P_i} \downarrow^\omega$. [11] and [12] give a completeness result based on 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 [5] we considered a three valued logic based on supervaluations, and showed it coincided with a proof procedure based on semantic tableaux allowing recursive calls. 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.

Research of Fitting partly supported by NSF Grant CCR-8702307 and PSC-CUNY Grant 667295.

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, 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, Partial models and logic programming, *Theoretical Computer Science*, vol 48, pp 229–255 (1986).

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

[7] M. C. Fitting, Logic programming on a topological bilattice, forthcoming in *Fundamenta Informaticae*.

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

[9] 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).

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

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

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

[13] 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).

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

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

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

[17] 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).