A Tutorial on the Well-Founded Semantics

Allen Van Gelder
Computer Science Dept., University of California
Santa Cruz, CA 95064 USA (e-mail: avg@cs.ucsc.edu)

October 26, 1997

Abstract

The well-founded semantics for "normal" logic programs is described using the alternating fixpoint construction as the basis. The presentation is informal. The semantics is motivated and illustrated through examples, and is compared briefly with earlier semantics for negation. Extensions to other forms of logic program are mentioned briefly. Recent work on practical implementations and applications is surveyed.

1 Introduction

The well-founded semantics [VGRS91] is a method for providing a static, or "declarative", meaning for a certain class of logic programs, called *normal* logic programs, in the terminology of Lloyd and Topor [LT84]. This paper provides a tutorial introduction to the well-founded semantics, using the alternating fixpoint construction (Section 3), which was shown to be equivalent to the original formulation [VG93a]. The main properties are described (Section 4), with emphasis on those that are relevant to implementations (Section 5). Extensions of the alternating fixpoint construction to other logical systems are also described briefly (Section 6).

1.1 Notation and Definitions

The standard terminology of logic and logic programming will be used as far as possible [Llo87]. We shall define *normal logic programs* in a top-down fashion, with a minimum of formality.

Definition 1.1: A *normal logic program* is a finite set of *normal rules*, or possibly an infinite set of *normal rules* that has a finite presentation. A *normal rule* is a syntactic construct of the form

$$head \texttt{ :- } body$$

where the symbol "`:-`" is usually read as "if". The *head* is an atomic formula. The *body* consists of a finite sequence of zero or more *subgoals*.

Each *subgoal* is a *literal*, which is defined as an atomic formula or a negated atomic formula. Negation in rules is denoted as "¬". If the rule body is empty, the symbol "`:-`" may be omitted; such a rule is often called a *unit clause*. A unit clause containing no variables is often called a *fact*. Following the Prolog convention, literals in the rule body are separated by commas, and the sequence is terminated with a period.

A *query* is like a rule with no head, syntactically "`:-` *body*". Usually, a query is not part of a logic program, but is defined at "run time", and defines the problem to be solved by a particular run of the logic program. Queries are sometimes called "goals", which are not to be confused with "subgoals". □

We assume a logical vocabulary of variables, function symbols, and predicate symbols. The number of arguments of a function symbol or predicate symbol, called its *arity*, is part of its identity. Function symbols of arity zero are called *constants*.

Terms and atomic formulas are defined in the standard way. With a fixed vocabulary a *term* is a variable or a function symbol with terms as arguments; an *atomic formula* (often abbreviated to *atom*) is a predicate symbol with terms as arguments. A *ground term* is a term containing no variables. A *ground atom* is an atomic formula whose arguments are ground terms.

By default, arguments of terms and atoms are written with the usual prefix functional notation, e.g., `s(0)`. However, we shall use infix notation for common arithmetic operators, as in Prolog, e.g., `X + 1`.

We shall use the ML convention for lists, treating them as terms built with the binary function symbol "::" (read as "cons") and the constant "`[]`" (read as "nil"). For easier readability, "::" is used as a right-associative operator. In addition, there is a square-bracket syntax for instantiated "lists", which is similar to Prolog. Thus, the term ::`(e,` ::`(+,` ::`(t, []))`) may be written as `e` :: `+` :: `t` :: `[]`, or, using the special notation of ML and Prolog, as `[e, +, t]`. Some familiar rules for lists illustrate the syntax:

$$\texttt{member(H, H :: T)} \quad . \tag{1}$$

$$\texttt{member(X, H :: T) :- member(X,T).} \tag{2}$$

$$\texttt{append([],L2,L2)} \quad . \tag{3}$$

$$\texttt{append(H :: T, L2, H :: L3) :- append(T,L2,L3).} \tag{4}$$

Numbers to the right are not part of the rules, but are reference numbers.

The scope of any variable is always the rule or query in which it occurs. That is, variables with the same name within a rule refer to the same variable, while variable names in separate rules always refer to distinct variables. Following the Prolog convention, variables begin with capital letters, while function symbols and predicate symbols begin with lowercase letters.

Literals and subgoals are called *positive* if they are atoms, and *negative* if they are negated atoms. A rule in which all subgoals are positive is called a *definite* rule. The term *Horn clause* is more general in that it may refer either to a definite rule, or to a query in which all subgoals are positive.

1.2 Extensional and Intentional Databases

It is customary to divide the predicate symbols of a logic program into two sets, the *extensional database* (EDB) and the *intentional database* (IDB). The EDB is given as a set of facts, or possibly unit clauses with variables; it is regarded as the input to the program. No new EDB facts can be derived, of course. It is always assumed that any EDB atoms that are not given explicitly are false.

The IDB consists of rules, which are considered the invariant part of the program. The point of view for declarative semantics is that the rules with IDB symbols in their heads define a mapping from the EDB facts to some set of "derivable" IDB facts. The notion of "derivable" depends on the semantics. Therefore, in describing a model (for two-valued logic), it is common to mention only which IDB atoms are true. If we are using three-valued logic, it is necessary to specify which IDB atoms are false, as well.

1.3 Herbrand Universe and Basis

For this paper, let some logical vocabulary be fixed, which includes vocabulary of the logic programs and queries discussed, plus some *auxiliary* function symbols that do not appear in any program or query. The set of all ground terms constructible with this vocabulary is called the *Herbrand universe*, denoted as $\mathbf{H}$. The set of all ground atoms constructible with this vocabulary is called the *Herbrand basis*, denoted as $\mathbf{B}$. These sets play an important role in declarative semantics.

Auxiliary symbols involve technicalities that are beyond the scope of this tutorial [VGRS91, VGS93]. Briefly, their purpose is to ensure that a formula $\forall X\, F(X)$ is not true just because it happens to be true for the terms that can be constructed with the function symbols of the current program. As an example, suppose the whole program is $\mathtt{p}(\mathtt{a})$. The Herbrand universe based on the vocabulary of the program alone is $\{\mathtt{a}\}$. However, we do not want $\forall X\, \mathtt{p}(X)$ to be true for this program because, if an unrelated fact $\mathtt{q}(\mathtt{b})$ were added, then $\forall X\, \mathtt{p}(X)$ would not be true. Essentially, we take the view that the program is always part of some larger universe.

2 Horn Programs

Horn programs are logic programs in which all rules are definite, so that there are no negative subgoals. Van Emden and Kowalski initiated the research in declarative semantics of logic programs by giving the declarative semantics for Horn programs with Horn queries [VEK76]. This consists of the minimum model in the Herbrand universe $\mathbf{H}$.

Here, we are describing an Herbrand model by the set of elements of the Herbrand basis $\mathbf{B}$ that are true in the model. A model is said to be *minimal* if no proper subset of its true basis elements describes a model. If there is only one minimal model, it is said to be *minimum*.

For a fixed query, the *answer set* for that query is exactly the set of ground substitutions for variables appearing in the query such that all subgoals of the query occur in the minimum model mentioned above.

Van Emden and Kowalski demonstrated several important properties of their semantics for Horn programs with Horn queries [VEK76]. First, the intersection of any set of models over the same structure is also a model; thus, a unique minimum model exists. This model consists of those atoms in the Herbrand basis $\mathbf{B}$ that *must be true* when each rule *head* `:-` *body* is interpreted as the (universally closed) logical formula $\forall(\mathit{body} \rightarrow \mathit{head})$.

Furthermore, the Herbrand basis is a sufficiently refined structure to capture all properties of the Horn program that can be represented as Horn queries. Finally, the minimum model is the least fixpoint of the logical consequence operator defined in Section 2.1, the closure ordinal for this fixpoint is ω (the least infinite ordinal), and so the question of membership in the model is recursively enumerable. With these attractive properties, there is little reason to consider alternatives, and the semantics of Van Emden and Kowalski is considered standard.

2.1 Logical Consequence Operator and Least Fixpoint

So far we have been careful not to call rules formulas. However, the logical consequence operator, to be defined next, essentially evaluates the *body* of a Horn rule as a conjunction of its atoms. An empty rule body is *true*.

Definition 2.1: The *logical consequence operator* $\mathbf{T}_P$, for a given Horn logic program P, maps a set of atoms $\mathbf{I}$ into a set of atoms $\mathbf{T}_P(\mathbf{I})$, where both are subsets of the Herbrand basis $\mathbf{B}$. The mapping is defined as follows. A ground atom q is in the set $\mathbf{T}_P(\mathbf{I})$ if and only if there is some rule in P, and some assignment to the variables of the rule such that the head of the rule is q and all atoms in the rule body, as instantiated by the assignment, are in $\mathbf{I}$. $\Box$

The operator $\mathbf{T}_P$ is monotonic. Here, and throughout the paper, ω denotes the least infinite ordinal. The sequence $\mathbf{I}_0 = \emptyset$, $\mathbf{I}_{\alpha+1} = \mathbf{T}_P(\mathbf{I}_\alpha)$, where α ranges over the natural numbers, increases to a limit, denoted as $\mathbf{T}_P^{\omega}(\emptyset)$, where

$$\mathbf{T}_P^{\omega}(\emptyset) = \bigcup_{\alpha<\omega} \mathbf{I}_\alpha \tag{5}$$

In addition, $\mathbf{T}_P^{\omega}(\emptyset)$ is a fixpoint of $\mathbf{T}_P$, and by Tarski's theorem, it is the least fixpoint [VEK76].

In general, any monotonic operator ϕ has a least fixpoint, by Tarski's theorem, but $\phi^{\omega}(\emptyset)$ may not be a fixpoint. In this case, the iteration must continue past ω [Mos74]. Accordingly, we define the sequence as $\mathbf{I}_0 = \emptyset$, $\mathbf{I}_{\alpha+1} = \phi(\mathbf{I}_\alpha)$, and for every limit ordinal β, we define

$$\mathbf{I}_\beta = \phi^{\beta}(\emptyset) = \bigcup_{\alpha<\beta} \mathbf{I}_\alpha \tag{6}$$

This tutorial will not have programs or examples where such transfinite iteration is necessary.

3 Overview of the Alternating Fixpoint

In defining the *alternating fixpoint* of a normal logic program, $\mathbf{P}_0$, we wish to build upon the well understood ideas of definite rules, monotonic operators, least fixpoints and minimum models. Therefore, we shall introduce a set of *dual* predicate symbols.

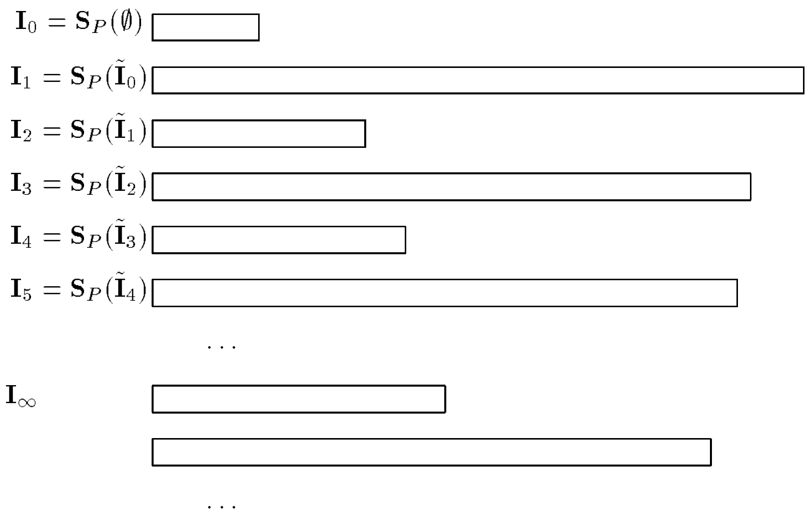

Figure 1: The sequence $\mathbf{I}_\alpha$ of fixpoints that defines the alternating fixpoint. Each $\tilde{\mathbf{I}}_\alpha = \tilde{\mathbf{B}} - (\sim \mathbf{I}_\alpha)$. By taking alternate terms of the entire sequence, one obtains either a monotonically increasing subsequence, called *underestimates* or a monotonically decreasing subsequence, called *overestimates*. The limit may occur at an ordinal greater than ω. The limiting underestimate, $\mathbf{I}_\infty$, coincides with the well-founded partial model.

Definition 3.1: Given a logic program, $\mathbf{P}_0$, for each predicate symbol $\mathtt{p_i}$, its *dual symbol* is denoted as $\tilde{\mathtt{p_i}}$. Now, for each rule in $\mathbf{P}_0$, we replace each negative subgoal on $\mathtt{p_i}$ by its positive version on $\tilde{\mathtt{p_i}}$. For example,

$$\mathtt{p_1(X) :- p_2(X,Y), \neg p_3(Y).} \qquad \text{becomes} \qquad \mathtt{p_1(X) :- p_2(X,Y), \tilde{p_3}(Y).}$$

Thus, the resulting program, denoted simply as $\mathbf{P}$, consists entirely of definite rules. When "$\sim$ appears as a prefix operator on a set of atoms, it denotes the dualization of each element. Thus, $(\sim \{p_1, \tilde{\mathtt{p_2}}\}) = \{\tilde{\mathtt{p_1}}, p_2\}$. $\square$

Since a dual symbol is never the head of a rule in $\mathbf{P}$, the only way to obtain dual facts is to supply them externally. We shall let $\tilde{\mathbf{I}}$ denote some set of dual facts that is taken as true. These facts are trivial definite rules, having empty bodies. The minimum model of $\mathbf{P} + \tilde{\mathbf{I}}$ exists, and will be denoted as $\mathbf{S}_P(\tilde{\mathbf{I}})$.

We are now ready to define the alternating fixpoint informally. We fix the Herbrand universe $\mathbf{H}$ and the Herbrand basis $\mathbf{B}$, as described in Section 1.3. Let $\tilde{\mathbf{B}}$ be $\mathbf{B}$ restricted to the dual predicate symbols.

We define a sequence of sets of atoms, $\mathbf{I}_\alpha$, as follows (see Figure 1). Let $\mathbf{I}_0$ be the minimum model of $\mathbf{P}$, which is clearly an underestimate of the facts that must be true. Define $\tilde{\mathbf{I}}_0 = \tilde{\mathbf{B}} - (\sim \mathbf{I}_0)$; this is clearly an *overestimate* of the dual facts that must be true (corresponding to facts on the original predicate symbols that must be false). (Observe that $\mathbf{I}_\alpha$ ranges over all predicate symbols, while $\tilde{\mathbf{I}}_\alpha$ ranges over only dual predicate symbols.)

But the above property of $\tilde{\mathbf{I}}_0$, together with monotonicity of $\mathbf{T}_P$, implies that the minimum model of $(\mathbf{P} + \tilde{\mathbf{I}}_0)$ must be an overestimate of all facts that must be true. Let $\mathbf{I}_1$ be this minimum model. Define $\tilde{\mathbf{I}}_1 = \tilde{\mathbf{B}} - (\sim \mathbf{I}_1)$; this is clearly an *underestimate* of the dual facts that must be true. Let $\mathbf{I}_2$ be the minimum model of $(\mathbf{P} + \tilde{\mathbf{I}}_1)$. By the same reasoning, $\tilde{\mathbf{I}}_2$ must again be an underestimate of all facts that must be true, as was $\mathbf{I}_0$. By reviewing the above steps, we see that we could have defined $\mathbf{I}_2$ as a transformation on $\mathbf{I}_0$ (with $\mathbf{P}$ as a fixed parameter).

Definition 3.2: Let $\mathbf{S}_P(\tilde{\mathbf{I}}_\alpha)$ denote the operation of constructing the minimum model of $(\mathbf{P} + \tilde{\mathbf{I}}_\alpha)$. We define:

$$\mathbf{A}_P(\mathbf{I}) = \mathbf{S}_P(\tilde{\mathbf{B}} - (\sim \mathbf{S}_P(\tilde{\mathbf{B}} - (\sim \mathbf{I})))) \tag{7}$$

□

The transformation $\mathbf{A}_P$ is easily seen to be monotonic, so it has a least fixpoint, which is called the *alternating fixpoint* of $\mathbf{P}$ [VG93a]. (The definition of $\mathbf{A}_P$ here varies slightly from the reference.) To see that $\mathbf{I}_\infty$ in Figure 1 is the least fixpoint of $\mathbf{A}_P$, observe that

$$\mathbf{A}_P^k(\emptyset) \subseteq \mathbf{A}_P^k(\mathbf{I}_0) \subseteq \mathbf{A}_P^{k+1}(\emptyset)$$

The main idea should be clear without getting involved in the many technicalities. One of these technicalities is that the closure ordinal for the alternating fixpoint may be greater than ω, the least infinite ordinal, even if we regard an application of $\mathbf{S}_P$ as a single operation. Despite this obstacle, we shall see that there are a number of useful applications for which queries can be answered with respect to the alternating fixpoint in finite time.

4 Comparison of Semantics for Negation

This section will summarize some earlier declarative semantics for normal logic programs with negation, and compare them with the well-founded semantics by means of examples.

As soon as negative subgoals enter the picture, the nice properties of Horn programs begin to disappear. If negation appears only in the query, then greatest fixpoints, as well as least fixpoints, need to be considered [AVE82]. But membership in the greatest fixpoint is neither recursively enumerable nor co-recursively enumerable. When negative subgoals may appear in rule bodies as well, the model-intersection property no longer holds. Moreover, the Herbrand universe of the program is no longer a satisfactory structure upon which to base a declarative semantics [JLL83]. Very technical issues arise, which are treated at some length in Lloyd's book [Llo87].

Example 4.1: This example shows that a simple minimal model semantics is not satisfactory. We wish to define rules that will find employees that have the maximum salary, based on a set of facts of the form `emp(Name, Sal)`, meaning that employee `Name` has salary `Sal`. We also assume the usual arithmetic comparison predicates are given.

The syntax restrictions of normal rules prevents us from expressing the maximum salary property, denoted as maxSal, in the simplest way: with a universally quantified first order formula in the rule body. Essentially, we would like to write a rule something like this:

$$\begin{aligned} \texttt{maxSal(Name, Sal) :- } & \texttt{emp(Name, Sal)} \wedge \\ & \forall \texttt{NS}\, (\texttt{emp(N, S)} \rightarrow (\texttt{S} \leq \texttt{Sal})) \end{aligned} \tag{8}$$

This rule can be transformed via DeMorgan's laws, replacing "not $\leq$" by "$>$", into:

$$\begin{aligned} \texttt{maxSal(Name, Sal) :- } & \texttt{emp(Name, Sal)} \wedge \\ & \neg\exists \texttt{NS}\, (\texttt{emp(N, S)} \wedge (\texttt{S} > \texttt{Sal})) \end{aligned} \tag{9}$$

which can now be split up into normal rules, with the aid of an intermediate predicate symbol, higherSal. The normal rules are:

$$\texttt{maxSal(Name, Sal) :- emp(Name, Sal), } \neg\, \texttt{higherSal(Sal).} \tag{10}$$

$$\texttt{higherSal(Sal) :- emp(N, S), S} > \texttt{Sal.} \tag{11}$$

This type of syntactic transformation is typical for expressing logical properties in the restricted syntax of normal rules.

The next point we wish to make with this example is that the minimum model idea that is used for Horn programs does not work here. Consider Rules 10–11 applied to the simple EDB of Figure 2. The expected model contains the true atoms higherSal(8), higherSal(10) and maxSal(vera, 12), and this model is minimal. However, there is another minimal model, which contains the true atoms higherSal(8), higherSal(10), and higherSal(12). Since the latter model does *not* contain maxSal(vera, 12), it is also a minimal model. □

4.1 Clark Program Completion

As Example 4.1 shows, the criterion of minimality fails immediately as a declarative semantics for rules with negation. Clark attempted to remedy this problem by introducing the concept of the *completed* program [Cla78]. The main idea was that rules are stated as "if" rules, but the programmer really intends that there is an implicit "only if" version as well.

Definition 4.1: Informally, the "only if" criterion is that a ground atom q must be false unless it is *forced* to be true in the following sense: (after the appropriate substitutions) q is the head of a rule for which all literals in the body are true. That is, all atoms in *positive* subgoals of the body are true and all atoms in *negative* subgoals of the body are false. See also Example 4.2. □

It is worth while to emphasize that the concept of minimal model is not explicitly present in Clark's completed-program semantics. With Horn programs, minimality meant minimal with respect to true atoms, because negated atoms simply were not an issue. Once negative subgoals appear, the program must be able to infer that certain atoms are false. Therefore, minimizing on true atoms is no longer appropriate.

```
emp(alex, 8).
emp(oleg, 10).
emp(vera, 12).
inDept(oleg, shoe).
inDept(alex, shoe).
inDept(vera, hat).
manages(oleg, shoe).
```

Figure 2: Extensional database (EDB) used in examples.

Returning to Example 4.1, there is no rule instance that forces `higherSal(12)` to be true, so it must be false. Thus, the "expected model" is the only model of the completed program. (More precisely, it is only Herbrand model for the specified vocabulary, but we shall always restrict attention to such models.)

4.2 Fitting and Kunen Three-Valued Semantics

In general, the completed program does not have a unique model. More troubling is the fact that it may have no models at all. For example, the one-rule program `p :-` $\neg$ `p` becomes an inconsistent completed program `p` $\longleftrightarrow \neg$ `p`, where "$\longleftrightarrow$" denotes "if and only if". Fitting pointed out this problem and numerous other anomalies that are even more paradoxical [Fit85]. He proposed a declarative semantics in three-valued logic. Kunen described a subtle variant [Kun87]. The main idea in both cases was to interpret the completed program in three-valued logic, with the third truth value being $\perp$ (undefined). The if-and-only-if operator has the special evaluation rule: $\perp \longleftrightarrow \perp$ is true. The practical effect is that the head of a rule may be undefined if the rule body evaluates as undefined. Then `p` is evaluated as $\perp$ in the program `p :-` $\neg$ `p`.

Although minimal two-valued models were not relevant for the Clark completed-program semantics, Fitting's semantics corresponds to the minimal three-valued model, which he shows always exists and is unique. The partial ordering among truth values in this case is: $\perp <$ *false*, $\perp <$ *true*, *true* and *false* are incomparable. Many technicalities are necessarily omitted from this brief review.

We can use the idea of *dual predicates* from Section 3 to sketch the construction of the Fitting minimum three-valued model. As in that section, negative subgoals in a program $\mathbf{P}_0$ are replaced by positive subgoals on dual predicate symbols, giving $\mathbf{P}$, which consists entirely of definite rules. Let $\mathbf{B}$ be the Herbrand basis of $\mathbf{P}$.

Definition 4.2: Given a set of atoms, $\mathbf{I} \subseteq \mathbf{B}$, we say a fully instantiated rule is *potentially usable* if none of its subgoals is "contradicted by" $\mathbf{I}$. In this context, atom p "contradicts" atom $\tilde{\mathrm{p}}$, and *vice versa*.

Let $\mathbf{I}_0 = \emptyset$. If $\mathbf{I}_\alpha$ has been computed, let $\mathbf{I}_{\alpha+1} = \phi(\mathbf{I}_\alpha)$, where $\phi(\mathbf{I}_\alpha)$ is defined as follows:

1. For any atom p on an *original* predicate symbol such that no rule with head p is potentially usable with respect to $\mathbf{I}_\alpha$, declare the corresponding $\tilde{\mathrm{p}}$ to be *true*; let $\mathbf{J}_\alpha$ be the set of all dual atoms declared true by this process.

2. $\phi(\mathbf{I}_\alpha) = \mathbf{T}_P(\mathbf{I}_\alpha \cup \mathbf{J}_\alpha)$.

For limit ordinals, Eq. 6 applies. □

The Fitting minimum model is the least fixpoint of the above operator ϕ. The Kunen semantics is based on $\phi^{\omega}(\emptyset)$, even if it is not a fixpoint. As mentioned before, no examples in this tutorial will involve ordinals greater than ω.

Example 4.2: This example prepares the way for Example 4.3, and also illustrates further the formation of the completed program. The object here is to define rules to specify that employees that are transitively managed by another employee, i.e., are "subordinates". The EDB is assumed to contain facts of the form seen in Figure 2. The IDB predicate, `subord(Mgr, Name, Sal)` means that `Name`, with salary `Sal`, is in the hierarchy managed by `Mgr`.

$$\mathtt{subord(Name, Name, Sal)\ :\text{-}\ emp(Name, Sal).} \tag{12}$$

$$\mathtt{subord(Mgr, Name, Sal)\ :\text{-}\ manages(Mgr, Dept),\ inDept(E, Dept),\ subord(E, Name, Sal).} \tag{13}$$

Notice that all employees are subordinates of themselves.

In forming the completed program, all rules with the same predicate symbol in the head are collected, and syntactically transformed if necessary, so that their heads unify. This step might require introducing equalities in the rule bodies, as seen below. Also, explicit quantifiers are added to each rule body for its local variables. Then the bodies are combined disjunctively. The completed form of Rules 12–13 is:

$$\begin{aligned} \mathtt{subord(Mgr, Name, Sal)} \longleftrightarrow\ & (\mathtt{Mgr} = \mathtt{Name}\ \wedge\ \mathtt{emp(Name, Sal)})\ \vee \\ & \exists \mathtt{Dept, E}\,(\mathtt{manages(Mgr, Dept)} \wedge \mathtt{inDept(E, Dept)} \wedge \mathtt{subord(E, Name, Sal)}) \end{aligned} \tag{14}$$

For Horn programs, such as this, the atoms that are true in the Fitting semantics and the Kunen semantics are precisely those in the minimum two-valued model of the original rules. For example, let us take the facts of Figure 2. Then the IDB atoms that are true are:

```
subord(vera, vera, 12)
subord(oleg, oleg, 10)
subord(alex, alex, 8)
subord(oleg, alex, 8)
```

In a minimum two-valued model, all other IDB facts would be false. Example 4.3 discusses their status in program-completion semantics. □

4.3 Stratified Programs

The database community (primarily) perceived problems with the Clark, Fitting, and Kunen approaches, all based on the "completed program" idea, because their semantics did not classify many atoms as false, although they were "obviously" false. As a further effect, other atoms were not classified as true. Specifically, when a set of recursive rules, such those of Example 4.2, were applied until a fixpoint was reached, atoms not derived in the process were not necessarily forced to be false. In the three-valued systems, they might be merely undefined. In the Clark system, they might be true in some models of the completed program, and false in others.

Example 4.3: Suppose, building upon Example 4.1 and Example 4.2, we desire rules to find those managers who have no subordinate in their hierarchy with a higher salary than their own. Let `bigMgr` be the IDB predicate for this property.

$$\texttt{bigMgr(Mgr) :- emp(Mgr, MgrSal)}, \neg\texttt{higherSubSal(Mgr, MgrSal)}. \tag{15}$$

$$\texttt{higherSubSal(Mgr, MgrSal) :- subord(Mgr, Sub, SubSal), SubSal} > \texttt{MgrSal}. \tag{16}$$

Looking at the facts given and derived in the previous examples, it seems that `bigMgr(oleg)` should be true, as the only other employee in the shoe department is `alex`, whose salary is 8. But surprisingly, there is a two-valued model of the completed program in which `subord(oleg, oleg, 12)` is true, `higherSubSal(oleg, 12)` follows from that, and `bigMgr(oleg)` is false.

In the minimal three-valued model, these atoms evaluate to $\perp$, because they are not forced to be false by the "only if" rules, and $\perp < \mathit{false}$ in the truth-value order. Therefore, `bigMgr(oleg)` is not true in any versions of the program-completion semantics.

It might seem that this behavior is an anomaly because we defined employees to be subordinates of themselves, making the relation reflexive. However, even if Rule 13 were modified to require `E` to be different from `Mgr`, a related problem would occur if the EDB contained the additional fact: `manages(alex, shoe)`. Then there is a two-valued model of the completed program in which the four facts:

`subord(alex, alex, 12)`   `subord(alex, oleg, 12)`   `subord(oleg, alex, 12)`   `subord(oleg, oleg, 12)`

are all true. □

The problem generally is that cycles of dependencies do not "fail finitely" in procedural terms. Prolog would go into infinite recursion in the previous example, given the goal `subord(oleg, oleg, 12)`. The program-completion semantics is consistent with this behavior; all versions say that this atom is neither true nor false. One goal of newer declarative semantics is to provide a framework for a more powerful logic programming language, rather than just characterize Prolog as it is.

The *stratified semantics* was the first to address the problem of cycles of positive dependencies [ABW88, VG89]. The stratified semantics is defined only for programs in which no cycles of rule dependencies involve negation. This is a syntactic condition.

Definition 4.3: Predicate symbol p is said to depend *immediately* on predicate symbol q if q appears in the body of some rule for p. The immediate dependency is negative if q appears in a negative subgoal. The dependency relation is the transitive closure of the immediate dependency relation.

A program is said to be *stratified* if the only negative dependencies are between predicate symbols in different strongly connected components (SCCs) of the dependency relation. □

When a program is stratified, the least fixpoints of the logical consequence operator in each SCC can be computed "bottom up", beginning with those whose rules have no negative subgoals. All atoms not in the least fixpoint of their SCC are declared false, those in the least fixpoints are true. Since a negative subgoal

```
prodn(e, [t], p01).                terminal(c).  % caret
prodn(e, [e, +, t], p02).          terminal(s).  % string
prodn(t, [s, a], p03).             terminal(u).  % underscore
prodn(t, [t, t], p04).
prodn(a, [b, p], p05).             nonterm(e).
prodn(a, [p, b], p06).             nonterm(t).
prodn(b, [], p07).                 nonterm(a).
prodn(b, [u, t, b], p08).          nonterm(p).
prodn(p, [p, c, t], p09).          nonterm(b).
prodn(p, [], p10).
                                   start(e).
```

Figure 3: A sample set of production rules for a context-free grammar, together with "facts" to specify terminal, nonterminal, and start symbols. Interpret "a" as "annotation", "b" as "subscripts", and "p" as "superscripts". Comments are introduced by a percent sign.

---

is only used after the least fixpoint of its SCC has been computed, its truth value is known. This ideal computation defines the *stratified model*.

The program of Examples 4.2 and 4.3 is stratified. For its stratified model, the true **subord** facts are computed completely first, as shown in Example 4.2. Those not found to be true are declared false, including **subord(oleg, oleg, 12)**. Then the true facts of **higherSubSal** are computed, and the rest declared false, including **higherSubSal(oleg, 10)**. Finally, the latter false fact enables **bigMgr(oleg)** to be derived.

It is straightforward to see that the alternating fixpoint computation on a stratified program produces the stratified model. Thus the well-founded semantics is a generalization of the stratified semantics to include unstratified programs. In the next section we shall look at unstratified programs in more detail.

4.4 Unstratified Negative Subgoals

To motivate the choice of the well-founded semantics for normal logic programs, let us examine a typical application in which unstratified negative subgoals arise. In this application we formulate rules to analyze a context free grammar (CFG), in preparation for generating a parse table for it. (See Aho, Sethi and Ullman for a review of parsing concepts and terminology [ASU85].) We assume the production rules of the CFG are given as a set of atomic formulas on the ternary symbol prodn. A rule of the form

$$\texttt{prodn(A, [}\texttt{B}_1, \ldots, \texttt{B}_\texttt{k}\texttt{], PN).} \qquad k \geq 0 \tag{17}$$

represents the production rule: $\texttt{A} \rightarrow \texttt{B}_1 \cdots \texttt{B}_\texttt{k}$. The third argument, PN, is a label or name for the production rule.

Recall that A is a nonterminal symbol, while $\texttt{B}_\texttt{i}$ are terminal or nonterminal symbols. To complete the specification of the CFG, additional predicate symbols are used to specify the start symbol (**start**), the nonterminal symbols (**nonterm**), and the terminal symbols (**terminal**). A sample expression grammar is shown in Figure 3. However, the grammar itself should be thought of as the IDB, i.e., *input* to the program, rather than part of it.

The concept of *nullable* symbols (also called *erasable* symbols) arises in the study of both formal and natural languages, typically in connection with optional words or phrases. One task of grammar analysis is to determine certain relationships, called `first` and `follow`, between nonterminal symbols and terminal symbols [ASU85], and their definitions involve *nullable* symbols.

Example 4.4: A nonterminal symbol is *nullable* if it can possibly derive the empty string (denoted as ϵ) in some finite number of steps. In words, we can say that symbol `V` is nullable if there is *some* production rule, $\mathtt{V} \rightarrow \mathtt{B_1} \cdots \mathtt{B_k}$, $k \geq 0$, such that all of the symbols $\mathtt{B_1}, \ldots, \mathtt{B_k}$ are nullable. We shall call this a *nullable* production rule. (See Eq. 19.)

The nullable production rule property, denoted as `nullProdn`, is most naturally expressed with a universal quantifier in the rule body:

$$\begin{aligned} \mathtt{nullProdn(V,Bs,PN)} \ \mathtt{:-}\ \mathtt{prodn(V,Bs,PN)} \ \wedge & \\ & \forall W \, (\mathtt{member}(W, \mathtt{Bs}) \rightarrow \mathtt{nullable}(W)) \end{aligned} \tag{18}$$

As in Example 4.1, a series of source transformations produces normal rules, introducing an intermediate predicate symbol, `extProdn`. The rule for `nullable` is also included.

$$\mathtt{nullable(V)} \ \mathtt{:-}\ \mathtt{nullProdn(V,Bs,PN)} \tag{19}$$

$$\mathtt{nullProdn(V,Bs,PN)} \ \mathtt{:-}\ \mathtt{prodn(V,Bs,PN)}, \neg\, \mathtt{extProdn(Bs)} \tag{20}$$

$$\mathtt{extProdn(Bs)} \ \mathtt{:-}\ \mathtt{member(W,Bs)}, \neg \mathtt{nullable(W)} \tag{21}$$

The rules for `member` were given in Rules (1–2).

With the production rules of Figure 3, we would expect "`b`", "`p`", and "`a`" to be nullable. We expect "`t`" and "`e`" to be not nullable. □

Notice that `nullable`, `nullProdn`, and `extProdn` form an SCC and have negative dependencies. Therefore the system is not stratified.

Next, let us trace the computation of Prolog for the query subgoal `nullable(a)`. As sketched in Figure 4, `nullable(b)` does succeed, but Prolog's attempt to derive `nullable(p)` gets stuck in an infinite recursion, using `p09`, because of its depth first search strategy. However, the atoms `nullable(p)` and `nullable(a)`, as well as `nullable(b)`, evaluate to *true* in the Fitting semantics and Kunen semantics. Thus these semantics capture the idea of finite failure under a robust search strategy, such as breadth first search, which is too inefficient for practical use in general.

In the construction of the three-valued minimum model, an atom that fails finitely under *some* search strategy will eventually have no potentially usable rules, as defined in Definition 4.2. At this point it is declared to be false (or its dual is true). Notation from Definition 4.2 is used below.

In this example, it is easy to see that `member(`W`, [])` has no potentially usable rules for any W, so its dual appears in $\mathbf{J}_0$, and it is false in $\mathbf{I}_1$. Then, `extProdn([])` has no potentially usable rules, and is false in $\mathbf{I}_2$. The atoms `nullProdn(p, [], p10)` and `nullable(p)` enter the model at $\mathbf{I}_3$ and $\mathbf{I}_4$, respectively. Similarly, `nullable(b)` is in $\mathbf{I}_4$. Now, `extProdn([b, p])` has no potentially usable rules with respect to $\mathbf{I}_4$,

```
:- nullable(a)?
:- nullProdn(a, Bs, PN)?
:- prodn(a, [b,p], p05), ¬ extProdn([b,p])?
:- ¬ extProdn([b,p])?
    :- extProdn([b,p])?
    :- member(W, [b,p]), ¬nullable(W)?
    :- ¬nullable(b)?
        :- nullable(b)?
        :- nullProdn(b, Bs, PN)?
        :- prodn(b, [], p07), ¬ extProdn([])?
        :- ¬ extProdn([])?
            :- extProdn([])?
            :- member(W, []), ¬nullable(W)?
            :- false.
        :- true.
    :- false
       (backtrack)
    :- member(W, [p]), ¬nullable(W)?
    :- ¬nullable(p)?
        :- nullable(p)?
        :- nullProdn(p, Bs, PN)?
        :- prodn(p, [p,c,t], p09), ¬ extProdn([p,c,t])?
        :- ¬ extProdn([p,c,t])?
            :- extProdn([p,c,t])?
            :- member(W, [p,c,t]), ¬nullable(W)?
            :- ¬nullable(p)?
               ...
```

Figure 4: Prolog cannot derive `nullable(a)` because the subgoal `nullable(p)` causes infinite recursion. However, a breadth-first strategy would permit `nullable(p)` to succeed and `extProdn([b, p])` to fail finitely, after which `nullable(a)` succeeds.

---

so its dual enters $\mathbf{J}_4$ and $\mathbf{I}_5$, `nullProdn(a, [b, p], p05)` enters at $\mathbf{I}_6$, and `nullable(a)` enters the model at $\mathbf{I}_7$.

As a general principle, when rules contain *no* positive dependency cycles, as in Example 4.4, the well-founded semantics, Fitting semantics, and Kunen semantics all agree. The exceptions to this general principle seem to consist of artificially composed programs, whose purpose is just to demonstrate differences among the semantics [VGS93].

In the alternating fixpoint construction (Section 3) the analog of finite failure is that the atom fails to be derived even in the overestimate. Intuitively, if an atom on an original predicate symbol has no potentially usable rules, as defined in Definition 4.2, it will not appear in the overestimate being constructed, and so its dual *will* appear in the next underestimate.

The difference from the Fitting and Kunen operators arises when there are unsupported cycles of positive dependencies (even p `:-` p). In this case the atoms in the unsupported cycle are never declared false because they do have potentially usable rules. However, they do *not* get derived in the overestimate of the alternating fixpoint construction, so their duals *do* appear in the subsequent underestimates. This was seen in Example 4.3.

For this example, `member(`W`, [])` does not appear in the overestimate $\mathbf{I}_1$, for any W, so `extProdn([])` is also absent from $\mathbf{I}_1$, and its dual appears in the underestimate $\mathbf{I}_2$. (The sets $\mathbf{I}_\alpha$ are now defined as in Section 3.) However, `extProdn([b, p])` *is* derived in the overestimate $\mathbf{I}_1$ because the duals of `nullable(b)` and `nullable(p)` are present.

Now, `nullProdn(p, [], p10)` and `nullable(p)` are derivable in $\mathbf{I}_2$. Similarly, `nullable(b)` is in $\mathbf{I}_2$. Therefore, their duals are *no longer* present in the next overestimate $\mathbf{I}_3$, so `extProdn([b, p])` cannot be derived in that overestimate. It follows that the dual of `extProdn([b, p])` appears in the underestimate $\mathbf{I}_4$, enabling `nullable(a)` to be derived.

The status of `nullable(t)` and `nullable(e)` in all three semantics is "undefined". Intuitively, the symbol `t` should not be nullable, based on the production rules of Figure 3, but `p04` prevents all of the fixpoint constructions seen so far from making progress. We shall return to this issue in Section 6.

5 Implementations and Applications

This section reviews some of the implementation efforts for the well-founded semantics, and mentions a few applications. There have been two main implementation approaches, one for function-free programs and one for more general programs. For function-free programs, the idea is to imitate the construction of the alternating fixpoint, but restrict the computation somehow to "relevant" atoms for a given query; this is essentially a "bottom-up" computation, which relies on the finiteness of the Herbrand basis to guarantee termination. For general programs, the idea is to strengthen the Prolog search mechanism, while working back from the query to the subgoals; this is essentially a "top-down" computation. The top-down method maintains a finite amount of information at each step, but may not terminate if the Herbrand basis is infinite.

5.1 Bottom-Up Methods for Function-Free Programs

Bottom-up approaches have been based on "magic sets" [BMSU86]. The idea of "magic sets" is to add a supplementary set of predicate symbols, called "magic predicates", to the program, whose purpose is to restrict the computation to relevant facts for a given query. The degree to which this is successful varies with the program rules and with the EDB. Original rules are modified to include additional "magic" subgoals, and rules for the "magic predicates" are added to the program. Implementations of the methods of this section have been reported only for function-free programs, and only at a "prototype" level.

"Magic predicates" interact recursively with original predicates. Ross observed that this interaction could change the well-founded model of the transformed program [Ros90]. He described a modification that worked correctly for *modularly stratified* programs [Ros90]. The class of *modularly stratified* programs is a generalization of stratified programs that requires additional constraints on the EDB to be assumed.

Kemp, Srivastava and Stuckey described an application of the magic set idea to the alternating fixpoint construction [KSS95]. They defined a rule to be *allowed* if each variable of the rule appears in some positive subgoal. They proved correctness for all *allowed* programs [KSS95].

Morishita described a novel way to combine the generation and use of "magic tuples" with the alternating fixpoint construction [Mor96]. The main idea is to generate "magic tuples" normally when computing an overestimate, then use that same set of "magic tuples" for computing the next underestimate. The sequence of underestimates no longer increases monotonically, so correctness is not trivial. He defined a rule to be *safe* if each variable that appears in a negative subgoal also appears in a positive subgoal, or in the head of the rule. He proved correctness for *safe* programs.

5.2 Top-Down Methods

An early top-down approach restricted proof searches to "tight" derivations [VG89]. It was designed to capture stratified negation (Section 4.3), but the method functioned on unstratified programs, as well. The idea of tight derivations underlies the newer top-down methods, and goes back to the Model Elimination theorem-proving method [Lov78]. The idea is simply that a shortest derivation will not contain any subgoal that is identical to an ancestor of itself. A derivation that does *not* have any such subgoals is called *tight*. Therefore, if such a subgoal is created, that branch of the search can be abandoned as failing. The more common occurrence is that the subgoal is not syntactically identical to an ancestor, but is a variant or a more specific subgoal. In this case, the search can be postponed, but not failed. Solutions discovered later at the ancestor may need to be "recycled" into the postponed node. However, no implementation of tight derivations was reported.

Researchers at Stony Brook, led by David S. Warren, reported on a series of logic-programming implementations, culminating in a system named *XSB* [CW93, CSW95, CW96b, SSW96]. This system is available via ftp and the internet, and appears to be the only "production quality" implementation of the well-founded semantics. It is based on a conceptual resolution strategy called *SLG*, The twin themes are *tabling* and *delays*. We will illustrate these ideas, but there are many technical details, for which the reader should consult the cited papers.

Consider again Rules 19–21 for the *nullable symbol* property, the example grammar (Figure 3), and the unsuccessful attempt of Prolog to derive `nullable(a)` (Figure 4).

Suppose SLG is given the query `nullable(`X`)`. It creates a table entry to record that this goal is active, then looks for rules with which to unify it, and finds Rule 19. Now it creates an entry noting that `nullProdn(`X`,` Bs`,` PN`)` is active, and locates Rule 20 with which it unifies. Then it creates entries for `prodn(`X`,` Bs`,` PN`)` and `extProdn(`Bs`)` with a notation that the latter is negative.

If the `prodn` facts in Figure 3 are processed in order, we will go through a number of steps involving goals `nullable(e)` and `nullable(t)`, which terminate without finding any solutions, and are omitted.

We pick up the computation when the fact `prodn(a, [b, p], p05)` is processed. This provides a *solution* for the active goal `prodn(`X`, `Bs`, `PN`)`. This solution involves a restricting unification, so a new, more specific, table entry is created for `extProdn([b, p])`, which is the remaining subgoal in the body of Rule 20. Should this new subgoal succeed or fail, the substitution will be passed back to the more general goal of which it is a special case.

Except for creating table entries, the SLG computation might proceed along the lines of Figure 4. However, whenever a new subgoal is encountered, the table is checked to see it is a variant of a goal that is already active. If so, then that existing table entry is annotated to furnish any solutions to this point in the derivation structure, and the derivation continues elsewhere. Also, when `nullProdn(p, `Bs`, `PN`)` is activated, it is recognized as a special case of the already active `nullProdn(`X`, `Bs`, `PN`)`. Therefore, when the first solution of `member(`W`, [p, c, t])` instantiates W to `p`, and the second `nullable(p)` goal occurs (see the end of Figure 4), it is associated with the same table entry as the first occurrence of `nullable(p)`, some seven lines above. Clearly, some very precise bookkeeping is needed!

Having recognized that `nullable(p)` is already an active subgoal, the new subgoal is *delayed*. The procedure looks for somewhere else to make progress, and finds a different solution for `member(`W`, [p, c, t])`. The new solution instantiates W to `c`, which is a terminal symbol, and is easily found not to be nullable. Now `extProdn([p, c, t])` succeeds, and `nullProdn(p, [p, c, t], p09)` fails. However, its parent subgoal, `nullProdn(p, `Bs`, `PN`)`, does not fail yet, because there is another solution for `prodn(p, `Bs`, `PN`)` that has not been processed, namely `prodn(p, [], p10)`. Indeed, this leads to the success of `nullable(p)`, along lines similar to `nullable(b)`. As the results are passed back, all attempts to derive `extProdn([b, p])` are found to have actually failed, not merely to have been delayed. Thus it reports failure, and `nullable(a)` succeeds.

On the other hand, attempts to derive `nullable(t)` lead to delayed nodes, rather than failures. When all attempts at progress are exhausted, the delayed nodes are inspected. If a strongly connected component of positive dependencies is found, all nodes in this SCC are evaluated as *false* simultaneously, and the computation resumes. However, in this example, each SCC involves some negative dependency, so all delayed nodes are evaluated as "undefined".

Again, we need to emphasize that SLG is a complex system, and this illustration left out many details. The papers need to be consulted to gain a thorough understanding.

5.3 Applications

The first report of an application that involved unstratified negation, where the well-founded model was needed, was a program to arbitrate simultaneous moves in the board game, *Diplomacy* [VG90]. This program implemented the well-founded model for the specific rules it needed to evaluate.

With the availability of *XSB* as described in Section 5.2, it became feasible to develop applications directly in the well-founded semantics. For most rules, such powerful machinery as XSB is not required. However, many natural problems will have sections where stratified or unstratified negation needs to be handled correctly. Stony Brook researchers have reported a parser for the PTQ grammar [War95], program analysis tools [DRW96], and a model checker for software verification [RRR+97].

Another kind of application is the support of a library, or toolkit, for other programs to use as needed. Van Gelder described the use of the well-founded semantics to support aggregation in the presence of recursion [VG93b]. Chen and Warren described a method to support the computation of stable models [CW96a].

6 Extensions of the Alternating Fixpoint

It is natural to consider the alternating fixpoint construction for more general classes of rules. This section will mention two directions that have been explored.

Przymusinska and Przymusinski have proposed a *stationary semantics* for default logic [PP94]. The main idea is that more general default formulas take the place of dual atoms (or negative literals) in the alternating fixpoint construction. Przymusinski has investigated further extensions to include autoepistemic logic, and disjunctive logic programs [Prz94, Prz95]. (Also, see elsewhere in this issue.)

Van Gelder defined *alternating fixpoint logic* as the same basic alternating fixpoint construction, but applied to rules in which the rule body is any first-order formula [VG93a]. He gave an example to show that the resulting semantics differed from the corresponding program with normal rules, with the point being that some universally quantified subformulas evaluate to false, but after transformation into normal rules, their counterparts evaluate as undefined. However, he did not suggest any practical approaches to implementation of this logic.

Chen has developed a method to implement a substantial class of programs in *alternating fixpoint logic* [Che95]. The method is an extension of the top-down SLG framework (Section 5.2). The key idea is the evaluation of universally quantified implicational subgoals.

Example 6.1: Let us illustrate Chen's method for evaluating the "ideal" rule for `nullProdn` (Rule 18), which is restated here:

$$\begin{aligned}\texttt{nullProdn(V,Bs,PN) :- prodn(V,Bs,PN)} \wedge \\ \forall W\,(\texttt{member}(W,\texttt{Bs}) \rightarrow \texttt{nullable}(W))\end{aligned} \tag{22}$$

When the implicational subformula is to be evaluated, `Bs` will be instantiated to a specific list. For the EDB of Figure 3, we will at some point encounter the goal `nullProdn(t, [t, t], p04)`, as a result of using the instantiated rule,

$$\texttt{nullable(t) :- nullProdn(t, [t,t], p04)} \tag{23}$$

Recall that the queries `nullable(t)` and `nullable(e)` evaluated as "undefined" in the well-founded semantics, although it seems intuitively that they should be "false".

Clearly, the implication in the rule body only needs to be evaluated for those values of W that make the antecedant true. If those values comprise a finite set (or multi-set), $\{\texttt{W}_1, \ldots, \texttt{W}_\texttt{k}\}$, then Chen's innovation is to create an "equivalent" rule in which the universally quantified subformula (with quantified variable W) is replaced by the k-way conjunction of the consequents, each one having W replaced by a different $\texttt{W}_\texttt{i}$.

In this example, the multi-set of solutions is simply $\{\mathtt{t}, \mathtt{t}\}$ (the solutions of `member(`W`, [t, t])`), so the created rule is:

$$\mathtt{nullProdn(t,\ [t,t],\ p04)\ :\text{-}\ prodn(t,\ [t,t],\ p04),} \qquad\qquad$$
$$\qquad\qquad \mathtt{nullable(t),\ nullable(t).} \tag{24}$$

What has been accomplished is that `nullProdn(t, [t, t], p04)` now depends *positively* on `nullable(t)`, instead of through two levels of negation. Rules 23–24 define a *positive* cycle, in which both subgoals get delayed by SLG. When the delayed cycle was resolved in the normal program, it involved negation, so all subgoals were evaluated as undefined. After the rule transformation, there is no negation, so they are evaluated as false. Thus the queries `nullable(t)` and `nullable(e)` evaluate as "false", which is correct for alternating fixpoint logic. □

The situation becomes much more complicated if the antecedant is recursive with the head of the rule.

7 Conclusion

We have described the well-founded semantics for normal logic programs in terms of the alternating fixpoint construction. We have shown, through examples, its relationship to earlier proposals for semantics of negation. We have sketched some of the ideas that have been used to implement this semantics, and have indicated some of its applications. Finally, we mentioned some extensions, based on the alternating fixpoint idea.

This paper is by no means a survey of all significant work related to the well-founded semantics. There are hundreds of paper on various aspects of the subject. We have selected a few topics to give the reader a taste, with emphasis on those directions that look like being of practical use.

Acknowledgements

This research was partially supported by NSF grant CCR-9505036.

References

[ABW88] K. R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Los Altos, CA, 1988.

[ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison Wesley, ISBN 0-201-10088-6, 1985.

[AVE82] K. R. Apt and M. H. Van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Fifth ACM Symposium on Principles of Database Systems*, pages 1–15, 1986.

[Che95] W. Chen. Query evaluation in deductive databases with alternating fixpoint semantics. *ACM Transactions on Database Systems*, 20(3):239–287, 1995.

[Cla78] K. L. Clark. Negation as failure. In Gallaire and Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[CSW95] W. Chen, T. Swift, and D. S. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–199, 1995.

[CW93] W. Chen and D. S. Warren. A goal-oriented approach to computing the well-founded semantics. *Journal of Logic Programming*, 17(2-4):279–300, 1993.

[CW96a] W. Chen and D. S. Warren. Computation of stable models and its integration with logical query processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.

[CW96b] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[DRW96] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems – a case study. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1996.

[Fit85] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[JLL83] J. Jaffar, J.-L. Lassez, and J. Lloyd. Completeness of the negation-as-failure rule. In *Int'l Joint Conf. on Artificial Intelligence*, pages 500–506, 1983.

[KSS95] D. B. Kemp, P. J. Stuckey, and D. Srivastava. Bottom-up evaluation and query optimization of well-founded models. *Theoretical Computer Science*, 146(1-2):145–184, 1995.

[Kun87] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(4):289–308, 1987.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, New York, 2nd edition, 1987.

[Lov78] D. W. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland, Amsterdam, 1978.

[LT84] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[Mor96] S. Morishita. An extension of Van Gelder's alternating fixpoint to magic programs. *Journal of Computer and System Sciences*, 52(3):506–521, 1996.

[Mos74] Y. N. Moschovakis. *Elementary Induction on Abstract Structures.* North-Holland, New York, 1974.

[PP94] H. Przymusinska and T. C. Przymusinski. Stationary default extensions. *Fundamenta Informaticae*, 21(1):67–87, 1994.

[Prz94] T. C. Przymusinski. Well-founded and stationary models of logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(3-4):141–187, 1994.

[Prz95] T. C. Przymusinski. Static semantics for normal and disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 14(2-4):323–357, 1995.

[Ros90] K. A. Ross. Modular stratification and magic sets for Datalog programs with negation. In *Ninth ACM Symposium on Principles of Database Systems*, pages 161–171, 1990.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. S. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of Conference on Automated Verification*, 1997.

[SSW96] K. F. Sagonas, T. Swift, and D. S. Warren. An abstract machine for computing the well-founded semantics. In *Proceedings of the Joint Conference and Symposium on Logic Programming*, Bonn, Germany, Sep 1996. MIT Press.

[VEK76] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[VG89] A. Van Gelder. Negation as failure using tight derivations for general logic programs. *Journal of Logic Programming*, 6(1):109–133, 1989. Preliminary versions appeared in *Third IEEE Symp. on Logic Programming* (1986), and *Foundations of Deductive Databases and Logic Programming*, J. Minker, ed., Morgan Kaufmann, 1988.

[VG90] A. Van Gelder. Modeling simultaneous events with default reasoning and tight derivations. *Journal of Logic Programming*, 8(1):41–52, 1990.

[VG93a] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.

[VG93b] A. Van Gelder. Foundations of aggregation in deductive databases. In *Third International Conference on Deductive and Object-Oriented Databases*, Phoenix, AZ, December 1993.

[VGRS91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991. Preliminary abstract appeared in 1988 (7th) ACM Symposium on Principles of Database Systems.

[VGS93] A. Van Gelder and J. S. Schlipf. Common-sense axiomatizations for logic programs. *Journal of Logic Programming*, 17(2-4):161–195, 1993.

[War95] D. S. Warren. Programming the PTQ grammar in XSB. In R. Ramakrishnan, editor, *Applications of Logic Databases*, chapter 10, pages 217–234. Kluwer Academic Publishers, 1995.