ELSEVIER

Theoretical Computer Science 278 (2002) 25–51

Theoretical Computer Science

www.elsevier.com/locate/tcs

Fixpoint semantics for logic programming a survey

Melvin Fitting[a, b, *]

[a] *Department of Mathematics and Computer Science, Lehman College (CUNY), Bronx, NY 10468, USA*

[b] *Departments of Computer Science, Philosophy, Mathematics, Graduate Center (CUNY), 365 Fifth Avenue, New York, NY 10036, USA*

Abstract

The variety of semantical approaches that have been invented for logic programs is quite broad, drawing on classical and many-valued logic, lattice theory, game theory, and topology. One source of this richness is the inherent non-monotonicity of its negation, something that does not have close parallels with the machinery of other programming paradigms. Nonetheless, much of the work on logic programming semantics seems to exist side by side with similar work done for imperative and functional programming, with relatively minimal contact between communities. In this paper we summarize one variety of approaches to the semantics of logic programs: that based on fixpoint theory. We do not attempt to cover much beyond this single area, which is already remarkably fruitful. We hope readers will see parallels with, and the divergences from the better known fixpoint treatments developed for other programming methodologies. © 2002 Elsevier Science B.V. All rights reserved.

1. Introduction

A logic program consists of formulas of logic, generally written using some special, restricted syntax. One 'runs' a logic program by asking it questions – queries – and it is determined, by executing a proof engine, whether or not these queries follow from the program. Queries may contain free variables, in which case the intention is to determine for what values of the variables the queries follow from the program.

The preceding is a very general description, with much room for maneuver. A particular choice of syntax can place serious restrictions on programs that can be written or queries that can be asked. The choice of logic was left open above. Classical first-order logic is an obvious candidate, but incompleteness results tell us that, while we may be able to determine a query does follow, we cannot in general tell that it does not, and

* Correspondence address. Department of Mathematics and Computer Science, Lehman College (CUNY), Bronx, NY 10468, USA.

E-mail address: fitting@alpha.lehman.cuny.edu (M. Fitting).

URL: comet.lehman.cuny.edu/fitting

this affects the treatment of negation. Consequently, subsystems of classical logic are of interest. In the other direction, richer systems that permit numeric constraints or allow additional operators, such as temporal ones, are also of much interest. And finally, what about a proof engine? Completeness, in the classical sense, may be less important than efficiency on ‘probable’ queries. Thus a rich variety of systems fit into the general paradigm of logic programming. *Prolog* is the most familiar logic programming system today, though others have been implemented and experimented with.

The general description of logic programming above makes it clear that it is related to database query languages. The machinery provided is richer than is customary in that community, however. One piece of machinery that is commonly available is *negation*. In a simple database language, negation is not a problem (except possibly for efficiency considerations). Either an item is in a database, or it is not, and these facts can be reported no matter what. But if a system is built on classical first-order logic, negation can be a serious issue. Prolog without negation can, in a precise sense, compute exactly the recursively enumerable relations. If negation is added we would expect to have complements of recursively enumerable relations as well, and we know this is impossible. Instead a weaker version of negation is used – negation as failure. One concludes *not* X if X is not a consequence.

Negation as failure is inherently non-monotonic. If X is not a consequence of a particular program, so that *not* X is a conclusion, then if X is added to the program, the conclusion *not* X must be withdrawn. Moreover, it is not decidable in general that something is *not* a consequence. As a result of these considerations, more than one version of negation has been investigated. *Non-monotonic logic* is now seen as a close relative of logic programming, and developments in either area tend to affect both.

Since logic programming involves both logic and programming, it should not be surprising that several varieties of semantics have been developed for it. Some follow the model-theoretic approach of formal logic, and some are more like the fixpoint approach originally developed for imperative and functional programming. There are also game-theoretic approaches. The overall range of proposed semantics is vast, and somewhat bewildering.

In this survey paper we will almost entirely confine the discussion of logic programming semantics to the fixpoint approach. We will try to emphasize similarities with semantics developed for other programming paradigms. We do not mean to be encyclopedic – by now it really would require an encyclopedia. We will confine things to developments that have been of particular interest to the author. Others will have their own story to tell.

2. Syntax

The simplest of logic programming syntaxes is that of *Horn clauses*. *Prolog* essentially uses these, plus negation, which we will consider later on.

An *atom*, which logicians call an *atomic formula*, is an expression of the form $R(t_1,\dots,t_n)$, where R is a relation symbol and $t_1,\dots,t_n$ are *terms* that are built up from constant symbols and variables, using function symbols. It is also convenient to allow *false* and *true* as atoms. A *literal* is an atom, or the negation of an atom. These are also called *positive* and *negative* literals. A literal is *ground* if it is what logicians call *closed*, containing no variables.

A *Horn clause* is a disjunction of literals with at most one of them positive. Suggestive notation for the Horn clause $A \vee \neg B_1 \vee \cdots \vee \neg B_n$ is $A \leftarrow B_1, \ldots, B_n$, and it is what we use from now on. In this, A is the *head* of the Horn clause, and $B_1, \ldots, B_n$ is the *body*. If the head is of the form $R(t_1, \ldots, t_n)$, the Horn clause is said to be *about* the relation symbol R. The Horn clause $\leftarrow B_1, \ldots, B_n$ is identified with $\mathit{false} \leftarrow B_1, \ldots, B_n$, and $A \leftarrow$ is identified with $A \leftarrow \mathit{true}$. A *program clause* is a Horn clause whose head is non-empty – it is allowed that the body be empty.

Free variables in a Horn clause are thought of as universally quantified. In particular, any variables in a clause body that do not occur in the clause head can be thought of as existentially quantified in the body, because $(\forall x)[A \leftarrow B_1, \ldots, B_n]$ and $A \leftarrow (\exists x)[B_1, \ldots, B_n]$ are equivalent if x does not occur in A.

A *logic program* is a finite set of program clauses. Think of its members as joined conjunctively.

In practice, *unification* plays an essential role in the logic engine of a logic programming system. But as a first approximation in trying to understand logic programs semantically it is common to suppress this, using the following device.

Definition 1. If $\mathscr{P}$ is a logic program, an associated set $\mathscr{P}^*$ is constructed as follows: first, put in $\mathscr{P}^*$ all ground instances of members of $\mathscr{P}$; second, if a clause $A \leftarrow$ with empty body occurs in $\mathscr{P}^*$, replace it with $A \leftarrow \mathit{true}$; finally, if the ground atom A is not the head of any member of $\mathscr{P}^*$, add $A \leftarrow \mathit{false}$.

$\mathscr{P}^*$ will generally be infinite. It is a convenient fiction that $\mathscr{P}^*$ will do as a substitute for $\mathscr{P}$ and issues of unification can be ignored. It is a practice we follow throughout this paper.

3. Classical semantics

The intention is that a logic program, when executed, should answer 'yes' to certain queries – think of a logic program as determining which queries are true. If a query is not true with respect to a logic program, we will take it as false, though program execution may not, in fact, be able to tell us that since logic programs can be used to represent the r.e. relations, which are not closed under complementation. For now, 'yes' is positive information and anything else is not. We will take up possible approaches to 'no' later on.

Definition 2. A *valuation* is a mapping v from the set of ground atoms to the set of classical truth values $\{\mathit{false}, \mathit{true}\}$, meeting the conditions that $v(\mathit{true}) = \mathit{true}$ and

$v(\mathit{false}) = \mathit{false}$. We will often refer to a valuation as a *two-valued*, or *classical* valuation, to distinguish it from other kinds introduced later on.

Note 1. It is common in the logic programming literature for a valuation to be a *set* of ground atoms, rather than a function. The connection is: identify the set S of ground atoms with the valuation v that is true on exactly the members of S. In this paper we consistently take valuations to be functions rather than sets.

The standard approach in logic programming is to take *false* as the default – for a query to be *true* a reason for it to be so should be implicit in the program. This manifests itself in two ways. On the one hand, a ground atom A that is never mentioned in a program should be assigned the value *false*, and we have incorporated this by explicitly adding $A \leftarrow \mathit{false}$ to $\mathscr{P}^*$. On the other hand, if a ground atom does appear in a program, and either truth value can consistently be assigned to it, we should prefer *false* to *true*. To ensure this, we minimize with respect to the following ordering.

Definition 3. The space $\{\mathit{false}, \mathit{true}\}$ is given the truth ordering $\mathit{false} <_t \mathit{true}$, with $x <_t y$ not holding in any other case. We use $\leqslant_t$ as usual for $<_t$ or $=$.

$$\begin{tikzcd}
\mathit{true} \arrow[d, dash, "\leqslant_t"] \\
\mathit{false}
\end{tikzcd}$$

This ordering is extended to valuations pointwise: $v_1 \leqslant_t v_2$ if and only if $v_1(A) \leqslant_t v_2(A)$ for all ground atoms A.

Of course this gives the space of truth values, and hence the space of valuations, the structure of a complete lattice.

Note 2. If S_1 and S_2 are the sets of ground atoms associated with the valuations v_1 and v_2, respectively, as described above, then $v_1 \leqslant_t v_2$ if and only if $S_1 \subseteq S_2$. Set inclusion is often used in the literature in place of $\leqslant_t$.

The standard model-theoretic semantics is now easy to describe. It comes from [41], though in fact it goes back to [37, 38] – recommended references are [21, 1, 8, 28]. Think of a program clause $A \leftarrow B_1, \ldots, B_n$ as another way of writing the logic formula $(B_1 \wedge \cdots \wedge B_n) \supset A$, and recall, any free variables are to be thought of as universally quantified.

Definition 4. A *model* for a logic program $\mathscr{P}$ is a classical first-order model in which each member of $\mathscr{P}$ is true. A *Herbrand model* for $\mathscr{P}$ is a model for $\mathscr{P}$ whose domain is the set of closed terms, with an interpretation that makes each term of the language designate itself in the model.

Recall the definition of $\mathscr{P}^*$, Definition 1. Any model for $\mathscr{P}$ is also a model for $\mathscr{P}^*$. For Herbrand models, the converse is true as well: a Herbrand model for $\mathscr{P}^*$ is a Herbrand model for $\mathscr{P}$. Herbrand models are particularly simple to work with, since the domain is fixed and we only need to specify atomic truth conditions. In effect, we can identify Herbrand models for $\mathscr{P}^*$ with valuations, which simplifies things considerably. This is what we do from now on – Herbrand models *are* valuations.

It can be shown that, among all Herbrand models for a given program there is a smallest with respect to the $\leqslant_t$ ordering of valuations. This supplies the standard semantics for the program, and it agrees well with the general intuition about logic programs and with the behavior of (idealized) Prolog. Moreover, several other approaches to logic program semantics have turned out to be equivalent to this one. It is quite firmly established.

Among all Herbrand models, *supported models* are singled out for special attention. Essentially, the idea is, if a ground atom is true in such a model it must not be "by accident," but rather some clause in the program should justify its truth. Here is one way of characterizing the notion rigorously, though it does take a detour through logic with infinitely long expressions.

Definition 5. Let $\mathscr{P}$ be a logic program and let $\mathscr{P}^*$ be as usual. In $\mathscr{P}^*$, replace each ground clause $A \leftarrow B_1,\dots,B_n$ with $A \leftarrow (B_1 \wedge \cdots \wedge B_n)$. Next, if there are several clauses in the resulting set having the same head, $A \leftarrow C_1$, $A \leftarrow C_2,\dots.$ replace them with $A \leftarrow (C_1 \vee C_2 \vee \cdots)$. Since there could be infinitely many members with the same head we may wind up with a countable disjunction, but the semantic behavior of such an item is unproblematic. Call the set that results $\mathscr{P}^{**}$. In $\mathscr{P}^{**}$ a ground atom A turns up as the head of exactly one member.

Herbrand models for $\mathscr{P}$ are Herbrand models for $\mathscr{P}^*$ are Herbrand models for $\mathscr{P}^{**}$, and conversely. Now, in $\mathscr{P}^{**}$, replace each occurrence of $\leftarrow$ by $\equiv$, logical equivalence. A *supported model* for $\mathscr{P}$ is a Herbrand model in which all these equivalences are true.

It is not hard to show that the smallest Herbrand model for a logic program $\mathscr{P}$ is, in fact, a supported model, and hence the smallest supported model.

4. Apt-van Emden-Kowalski semantics

We turn to the central topic of this paper – fixed point approaches. We want to think of a logic program as a kind of 'revision operator'. If a program contains a clause $Q \leftarrow P$ and we believe P to be the case, this clause should force us to revise our beliefs so that Q is added to them (if it was not already there). The following *single-step* operator, from [2], is intended to capture one pass of such a revision.

Definition 6. Let $\mathscr{P}$ be a logic program. An associated mapping $T_{\mathscr{P}}$, from valuations to valuations, is defined as follows:

$$T_{\mathscr{P}}(v) = w,$$

where w is the unique valuation determined by the following: for a ground atom A,

(i) $w(A) = true$ if there is a ground clause $A \leftarrow B_1, \ldots, B_n$ in $\mathscr{P}^*$ with head A such that $v(B_1) = true$, and $\ldots$, and $v(B_n) = true$.

(ii) $w(A) = false$ otherwise.

Less formally, this says the following. $T_{\mathscr{P}}(v)$ makes a ground atom A true just in case A is the head of a ground instance of some clause in $\mathscr{P}$, and v makes the body of that ground instance *true*.

What we want is a valuation that the program cannot revise away – a fixed point for the single-step operator. And traditional lattice-theoretic arguments supply us such. The following has a straightforward proof.

Proposition 7. *For any program $\mathscr{P}$ the associated operator, $T_{\mathscr{P}}$, is monotone, that is, $v_1 \leqslant_t v_2$ implies $T_{\mathscr{P}}(v_1) \leqslant_t T_{\mathscr{P}}(v_2)$.*

Now the familiar *Knaster–Tarski Theorem*, [39], says single-step operators have smallest (and largest) fixed points. The smallest fixed point of $T_{\mathscr{P}}$ coincides with the smallest Herbrand model of the previous section, and thus supplies the standard semantics for $\mathscr{P}$. We will call this the *Apt-van Emden–Kowalski semantics* [2, 41].

More generally, the valuations v such that $T_{\mathscr{P}}(v) = v$ are the supported models for the program $\mathscr{P}$, while Herbrand models that are not necessarily supported are those v such that $T_{\mathscr{P}}(v) \leqslant_t v$, the *pre-fixed points* of $T_{\mathscr{P}}$.

Example 4.1. Here is a typical example. We will return to it or to variants of it from time to time. It is intended to recognize the even numbers, and incidentally, the odd numbers. Numbers are represented as numerals, using the constant symbol 0 and a successor function symbol s – thus $s^n(0)$, where we have n occurrences of s, represents the integer n:

$$\begin{aligned}
&even(0) \leftarrow,\\
&even(s(x)) \leftarrow odd(x),\\
&odd(s(x)) \leftarrow even(x).
\end{aligned}$$

If $\mathscr{P}$ is this program, $T_{\mathscr{P}}$ has a *unique* fixed point v, and $v(even(s^n(0))) = true$ if and only if n is even. Analogously for the odd numbers.

If we, somewhat artificially, add

$$even(x) \leftarrow even(x)$$

to the program above, the resulting single-step operator has the same smallest fixed point, but now the largest fixed point w is different: $w(even(s^n(0))) = true$ for all n, and $w(odd(s^n(0))) = true$ for all n except 0.

One way of proving the Knaster–Tarski Theorem is by approximation to the smallest and biggest fixed points. In the logic programming literature certain notation has become standard here. In presenting this we use *false* and *true* for the identically *false* and identically *true* valuations, respectively. Also, α is an arbitrary ordinal, and λ is an arbitrary limit ordinal:

$$
\begin{aligned}
&T_{\mathscr{P}}\uparrow_0 = \mathit{false},\\
&T_{\mathscr{P}}\uparrow_{\alpha+1} = T_{\mathscr{P}}(T_{\mathscr{P}}\uparrow_\alpha),\\
&T_{\mathscr{P}}\uparrow_\lambda = \bigvee\{T_{\mathscr{P}}\uparrow_\alpha \mid \alpha < \lambda\},\\
&T_{\mathscr{P}}\downarrow_0 = \mathit{true},\\
&T_{\mathscr{P}}\downarrow_{\alpha+1} = T_{\mathscr{P}}(T_{\mathscr{P}}\downarrow_\alpha),\\
&T_{\mathscr{P}}\downarrow_\lambda = \bigwedge\{T_{\mathscr{P}}\downarrow_\alpha \mid \alpha < \lambda\}.
\end{aligned}
$$

One shows the sequence $T_{\mathscr{P}}\uparrow_\alpha$ converges to the smallest fixed point of $T_{\mathscr{P}}$, and $T_{\mathscr{P}}\downarrow_\alpha$ converges to the biggest fixed point.

It is straightforward to show that $T_{\mathscr{P}}\uparrow_\alpha$ must reach the smallest fixed point by $\alpha = \omega$ – thus the smallest fixed point can have computational significance. It is also at the heart of proofs that the least fixed-point semantics agrees with many other semantical approaches that have been proposed for logic programs. On the other hand, the sequence $T_{\mathscr{P}}\downarrow_\alpha$ can be much more poorly behaved.

Example 4.2. Consider the following program:

$$
\begin{aligned}
&p(s(x)) \leftarrow p(x),\\
&q(0) \leftarrow p(x).
\end{aligned}
$$

The smallest fixed-point valuation is, in fact, $T_{\mathscr{P}}\uparrow_0$, and maps every ground atom to *false*. This is also the biggest fixed point, but the downward approximation sequence does not settle on this valuation until stage $\omega + 1$.

Much worse behavior is possible and in general the downward approximation sequence may not attain a constant value before Church–Kleene ω_1. Unfortunately, this bad behavior of the downward approximation sequence becomes an important problem once an enrichment of the logic programming machinery is attempted.

5. Negation

Perhaps the most desirable, and most controversial, addition to the basic logic programming machinery is negation. We have already remarked that, since logic programs give us exactly the r.e. relations, a real classical negation cannot be added. Prolog adds what is called *negation as failure* – conclude *not* X if an attempt to establish X fails. Computationally, this too has its problems, since there may be infinitely many possible derivations of X to be explored. Negation as *finite* failure has been proposed as a more reasonable substitute – loosely, conclude *not* X if there are a finite number of possible ways to establish X, and all fail. One must understand, Prolog includes a version of negation that is operationally well-defined. The problem is to make semantic sense of it. And one must accept that when stated in this generality, the problem may not be solvable. It may be that Prolog's negation has a simple, intuitive meaning only for certain programs. At any rate, we will continue with our convenient habit of introducing simplifying assumptions, in order to get somewhere at all.

As a first approach, consider allowing negative (ground) queries, but do not allow negation to appear in programs themselves. In effect, keep the same notion of logic program that we used above, but require meaningful 'yes' and 'no' answers to queries.

It is here that the downward approximation sequence comes into play. It has been shown that, for a computationally meaningful notion of finite failure, the semantic counterpart is $T_{\mathscr{P}}\downarrow_{\omega}$ [7, 2]. That is, a no answer should be given to a query Q if $T_{\mathscr{P}}\downarrow_{\omega}(Q) = \mathit{false}$. Unfortunately, $T_{\mathscr{P}}\downarrow_{\omega}$ is not generally a fixed point of $T_{\mathscr{P}}$. Better behaved, semantically, is the largest fixed point of $T_{\mathscr{P}}$, which is $T_{\mathscr{P}}\downarrow_{\alpha}$ for some ordinal α that can be strictly bigger than ω. This too corresponds to an interesting notion of negation, but one that is generally not computable – the set of ground atoms falsified in the biggest fixed point of $T_{\mathscr{P}}$ can be Π^1_1 complete [5]. We thus have a choice between unnatural semantics, or uncomputable semantics. We choose to work with the largest fixed point of $T_{\mathscr{P}}$, rather than $T_{\mathscr{P}}\downarrow_{\omega}$, and hope that we can confine our attention to programs for which non-computability issues do not arise.

Allowing negative queries, but keeping unchanged the notion of logic program, does not go far enough. It is desirable to allow negation to enter into programs themselves.

Definition 8. A *general* program clause is an expression of the form

$$A \leftarrow L_1, \ldots, L_n,$$

where $L_1, \ldots, L_n$ are literals and not just atoms, and A is an atom. The body of a general program clause can be empty. A *general logic program* is a finite set of general program clauses.

Note that every logic program is also a general logic program. We continue our practice of working with the family of ground instances of the members of general logic programs, instead of explicitly invoking unification. The definition of $\mathscr{P}^*$ extends to general logic programs directly, Definition 1. We recognize that this avoids some fundamental issues.

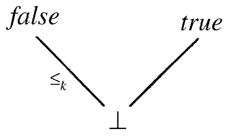
If $\mathscr{P}$ is a general logic program, the definition of the single-step $T_{\mathscr{P}}$ operator can be extended to cover it in a straightforward way – essentially we require that $v(\mathit{not}\ X)$ have the truth value $\neg v(X)$, and then keep the wording of Definition 6 intact. Unfortunately, this does not yield an adequate treatment. For one thing, the existence of smallest and biggest fixed points is no longer guaranteed since the presence of negations destroys monotonicity. What is worse, it is easy to see that for so simple a program as $P \leftarrow not\ P$, no fixed point exists at all. The approach that worked so well for logic programs without negation clearly needs some modification.

A guide to modification is, in fact, in front of us. For a logic program (without negations), *two* fixed points, not one, play a role – the smallest and the biggest of the fixed points of the associated single-step operator. This suggests the introduction of a *partial* valuation: if the two extreme fixed points agree on a classical truth value for the ground atom A, take that to be the value of A, and otherwise the value of A is *undefined*, or $\perp$. (See [6] for the logic background.) Consider the following extremely simple logic program $\mathscr{P}$: $P \leftarrow P$. If v is the smallest fixed point of $T_{\mathscr{P}}$ and V is the biggest, we have $v(P) = \mathit{false}$ but $V(P) = \mathit{true}$. Then, in the partial valuation semantics just proposed, we should take P to have $\perp$ as its value, with respect to program $\mathscr{P}$. Of course, this is different than the value assigned by the Apt-van Emden–Kowalski semantics, under which P is *false*. If we are to move towards such a partial semantics, then, we need compensating advantages to offset such shifts in what has come to be standard.

Definition 9. A *partial valuation* is a mapping v from the set of ground atoms to the set $\{\perp, \mathit{false}, \mathit{true}\}$, meeting the conditions $v(\mathit{false}) = \mathit{false}$ and $v(\mathit{true}) = \mathit{true}$. We often refer to partial valuations as *three-valued*.

This time we want $\perp$ to be the default, not *false*.

Definition 10. The space $\{\perp, \mathit{false}, \mathit{true}\}$ is given a knowledge ordering $\perp <_{\mathrm{k}} \mathit{false}$, $\perp <_{\mathrm{k}} \mathit{true}$, with $x <_{\mathrm{k}} y$ not holding in any other case. Then $\leqslant_{\mathrm{k}}$ is defined as usual:



The ordering is again extended to partial valuations pointwise: $v_1 \leqslant_{\mathrm{k}} v_2$ if and only if $v_1(A) \leqslant_{\mathrm{k}} v_2(A)$ for all ground atoms A.

This time the truth values do not give us a complete lattice, but we do have a cpo (in fact, a complete semi-lattice) that is quite familiar. The space of partial valuations inherits these algebraic features.

Note 3. Once again it is common in the literature to work with *sets* rather than mappings. A partial valuation is often represented by a disjoint pair of sets: $\langle T, F\rangle$, $T \cap F = \emptyset$. This corresponds to the partial valuation v that maps members of T to *true*, members of F to *false*, and members of neither to $\perp$. If this representation is used, one sets $\langle T_1, F_1\rangle \leqslant_{\mathrm{k}} \langle T_2, F_2\rangle$ if $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$. We find taking valuations as mappings to be much more convenient, and do so here.

A new single-step operator is associated with a general logic program $\mathscr{P}$, usually denoted $\Phi_{\mathscr{P}}$. In Definition 6 for $T_{\mathscr{P}}$, it was specified when an output valuation assigned *true*, and if it did not, *false* was the default. Now we explicitly specify when both *true* and *false* are assigned, and if neither is, $\perp$ is the default.

Definition 11. Let $\mathscr{P}$ be a general program. An associated mapping $\Phi_{\mathscr{P}}$, from partial valuations to partial valuations, is defined as follows:

$$\Phi_{\mathscr{P}}(v) = w,$$

where w is the unique partial valuation determined by the following: for a ground atom A,

(i) $w(A) = true$ if there is a general ground clause $A \leftarrow B_1, \ldots, B_n$ in $\mathscr{P}^*$ with head A, such that $v(B_1) = true$, and $\ldots$, and $v(B_n) = true$.
(ii) $w(A) = false$ if, for every general ground clause $A \leftarrow B_1, \ldots, B_n$ in $\mathscr{P}^*$ with head A, $v(B_1) = false$, or $\ldots$, or $v(B_n) = false$.
(iii) $w(A) = \perp$ otherwise.

Here is a suggestive alternate characterization of both $T_{\mathscr{P}}$ and $\Phi_{\mathscr{P}}$. Recall Definition 5 of $\mathscr{P}^{**}$ for logic programs – it extends directly to general logic programs, and we assume this in what follows. In $\mathscr{P}^{**}$ each ground atom occurs as the head of exactly one member.

If v is a classical, two-valued valuation, it extends to conjunctions, disjunctions, and negations in the usual truth-functional way. Likewise, if v is a *partial* or *three-valued* valuation, we can still extend v to disjunctions, conjunctions, and negations, but we must pick which three-valued logic we will be using. We choose Kleene's strong three-valued logic [22]. This is briefly described as follows. Negation switches *false* and *true*, and leaves $\perp$ unchanged. A conjunction is *true* if all its conjuncts are *true*; *false* if some conjunct is *false*, and $\perp$ otherwise. Disjunction is dual.

Now, here are the alternate characterizations we promised.

Two-valued: $T_{\mathscr{P}}(v) = w$, where w is the unique valuation determined by the following: if $A \leftarrow B$ is in $\mathscr{P}^{**}$, $w(A) = v(B)$ (where we use classical logic to evaluate $v(B)$).

Three-valued: $\Phi_{\mathscr{P}}(v) = w$, where w is the unique valuation determined by the following: if $A \leftarrow B$ is in $\mathscr{P}^{**}$, $w(A) = v(B)$ (where we use Kleene's strong three-valued logic to evaluate $v(B)$).

Of course when stated in this alternate form, generalizations are more easily suggested, as we will see later. What is crucial is that appropriate monotonicity conditions hold, and for partial valuations and the $\Phi_{\mathscr{P}}$ operator, this is the case.

Proposition 12. *For a general program $\mathscr{P}$, the operator $\Phi_{\mathscr{P}}$ is monotone with respect to $\leqslant_k$: $v_1 \leqslant_k v_2$ implies $\Phi_{\mathscr{P}}(v_1) \leqslant_k \Phi_{\mathscr{P}}(v_2)$.*

Since we do not have a complete lattice this time, the Knaster–Tarski theorem does not hold. Nonetheless, the algebraic structure is rich enough to ensure that monotone maps have smallest fixed points (though not biggest). The smallest fixed point of $\Phi_{\mathscr{P}}$

supplies what is sometimes called the *Kripke–Kleene semantics* for a general logic program $\mathscr{P}$ (and occasionally the *Fitting semantics*), [10], and also see [11, 14]. The use of the name Kleene is obvious – Kleene's strong three-valued logic is involved. Kripke's name is less obvious, but in fact there are close similarities between this semantics and a treatment of truth for sentences allowing self reference due to Kripke [23]. Indeed, the underlying mathematics is identical.

The earlier 'uparrow' and 'downarrow' notation that was used for the $T_{\mathscr{P}}$ operator is partly carried over to the present setting. Let us use $\perp$ for the partial valuation that is identically $\perp$ on all ground atoms:

$$\begin{aligned}
&\Phi_{\mathscr{P}}\uparrow_0 = \perp,\\
&\Phi_{\mathscr{P}}\uparrow_{\alpha+1} = \Phi_{\mathscr{P}}(\Phi_{\mathscr{P}}\uparrow_\alpha),\\
&\Phi_{\mathscr{P}}\uparrow_\lambda = \bigvee\{\Phi_{\mathscr{P}}\uparrow_\alpha \mid \alpha < \lambda\}.
\end{aligned}$$

Here, as before, λ is an arbitrary limit ordinal, but now the least upper bound operation is with respect to the $\leqslant_k$ ordering rather than the $\leqslant_t$ ordering. There is no corresponding 'downarrow' version, since there is no lattice top at which to start. Still, the sequence $\Phi_{\mathscr{P}}\uparrow_\alpha$ converges to the least fixed point of $\Phi_{\mathscr{P}}$, as expected.

If $\mathscr{P}$ is a logic program, not a general logic program, i.e., if it does not involve negation, then both the Apt-van Emden–Kowalski semantics and the Kripke–Kleene semantics apply, and they are *not* in general the same. The simple program $P \leftarrow P$ is a good example. In the Apt-van Emden–Kowalski semantics for this program, P receives the value *false*, but in the Kripke–Kleene semantics, P receives the value $\perp$. Which is the 'right' choice? A good case can be made for either. A value of *false* is reasonable because it does not follow from the information in the program that P should be *true*, so it should be taken to be *false* by default. On the other hand, the program actually gives us no usable information about P whatsoever so we could say that, as far as this program is concerned, P should be $\perp$ – no information. It does not seem possible to choose between these on any 'intrinsic' grounds. Intended applications probably should decide.

Even though the two semantics differ on programs without negations, they are not unrelated. As we noted earlier, both the smallest and the biggest fixed points of $T_{\mathscr{P}}$ play a natural role. In fact, the role of these two is folded into the smallest fixed point of $\Phi_{\mathscr{P}}$ rather neatly.

Proposition 13. *Let $\mathscr{P}$ be a logic program* (*without negations*). *Also*, *let v_k be the smallest fixed point of $\Phi_{\mathscr{P}}$* (*with respect to $\leqslant_k$*), *and let v_t and V_t be the smallest and the biggest fixed points of $T_{\mathscr{P}}$* (*with respect to $\leqslant_t$*). *Then*, *for a ground atom A*,
(i) *If $v_t(A) = V_t(A)$, then $v_k(A)$ has this common value.*
(ii) *If $v_t(A) \neq V_t(A)$, then $v_k(A) = \perp$.*

There are some unfortunate side effects of this otherwise nice-looking proposition. Since $T_{\mathscr{P}}\downarrow_\alpha$ may need Church–Kleene ω_1 steps to converge to the largest fixed point of

$T_{\mathscr{P}}$ it follows (with a little argument) that $\Phi_{\mathscr{P}}\uparrow_{\alpha}$ may need as many steps to converge to the smallest fixed point of $\Phi_{\mathscr{P}}$. Likewise, there are programs for which the least fixed point of $\Phi_{\mathscr{P}}$ is Π_1^1 complete. Logic programming is one of the rare programming paradigms where such non-continuity issues arise naturally. Perhaps rather than devising more a complex semantics, we should impose syntactic restrictions that tell us, "don't write that program".

Howard Blair takes a somewhat different, and quite interesting, position. In a personal communication he writes, "that is just the sort of program you want to write when you want to write something that deliberately exhibits the complexity of nondeterministic computational processes that emphasize fairness or infinitely branching nondeterminism. Both have associated Π_1^1 complete decision problems. For example, in the case of infinitely branching nondeterminism, there is a strong halting problem (does the process halt on all computation paths?) which is Π_1^1 complete, and this latter decision problem is the source of the Π_1^1 completeness of the Herbrand rule's failure set."

On the other hand, when it comes to logic programs with negations, the Kripke–Kleene semantics wins by default. Since such programs do not give us monotonic $T_{\mathscr{P}}$ operators, the Apt-van Emden–Kowalski approach simply does not apply. As a simple example, consider the program consisting of $P \leftarrow not\ P$. With only the classical two truth values available, a single-step operator would never settle on a value for P – we would have a period two oscillation. But the Kripke–Kleene semantics simply assigns P the value $\bot$, as one would expect. Here is another, more complex example, modifying Example 4.1.

Example 5.1. Once again, numbers are represented as numerals, using a constant symbol 0 and a successor function symbol s. Let $\mathscr{P}$ be the following program:

$$even(0) \leftarrow,$$

$$even(s(x)) \leftarrow not\ even(x).$$

(This program even behaves properly in Prolog.) The least fixed point of $\Phi_{\mathscr{P}}$ assigns $even(t)$ the value *true* if t is a numeral naming an even number, and assigns it *false* if t names an odd number. The value $\bot$ is never assigned – we actually have a two-valued map here. In fact, for this program, $\Phi_{\mathscr{P}}\uparrow_{\alpha}$ reaches its smallest fixed point in ω steps. All this good behavior will be of significance later on.

In order to deal with some of the problems of the Kripke–Kleene semantics mentioned above, Kunen introduced a modification [24–27]. Suppose we cut off the approximation sequence at ω – but this must be done carefully, it *does not* mean working with $\Phi_{\mathscr{P}}\uparrow_{\omega}$. Here is what Kunen showed.

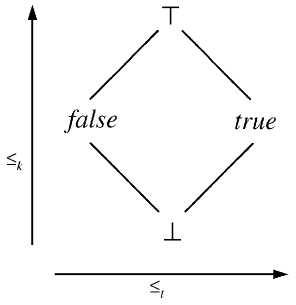
Proposition 14. *There is a model M such that, for a ground sentence A, A is true in M if and only if A is true in $\Phi_{\mathscr{P}}\uparrow_n$ for some integer $n<\omega$. Moreover, truth in M is recursively enumerable.*

This provides us with quite a plausible semantics for $\mathscr{P}$, even though it is not quite a fixed-point semantics. But it is important to understand what is not being said here. If $P(x)$ is atomic, it could happen that, for each ground instance $P(t)$, there could be some integer n such that $P(t)$ is true in $\Phi_{\mathscr{P}}\uparrow_n$. But n may depend on t, so for no integer n would *every* instance be true in $\Phi_{\mathscr{P}}\uparrow_n$, and hence the universal quantification of $P(x)$ would not be true in any $\Phi_{\mathscr{P}}\uparrow_n$ either, and thus not in M, although we would expect it to be true in $\Phi_{\mathscr{P}}\uparrow_\omega$. To bring this about, M must be non-Herbrand – its domain must consist of more than just closed terms of the language. This gives it a flavor that makes it difficult to use in practice, though it has proved useful for theoretical investigations.

6. Belnap's logic

The ordering $\leqslant_k$ on Kleene's three truth values does not give us a complete lattice. This may or may not be seen as a disadvantage, but it does suggest investigating what happens when a top is added. Belnap [4] introduced a four-valued logic that extends that of Kleene, with the explicit intention of providing a logic in which inconsistencies can be represented without everything becoming a consequence. It turns out that Belnap's logic allows inconsistencies to appear in logic programs in a useful way as well. And it has other consequences of considerable interest, as we discuss below.

Belnap observed that his four truth values have *two* natural orderings. Both are shown in the following diagram:



The vertical knowledge ordering $\leqslant_k$ extends the one we were using with the three truth values of Kleene. Belnap's four truth values can be thought of as *sets* of ordinary truth values, so that $\bot = \emptyset$, $\mathit{false} = \{\mathit{false}\}$, $\mathit{true} = \{\mathit{true}\}$, and $\top = \{\mathit{false}, \mathit{true}\}$. If we do this, $\leqslant_k$ becomes simply $\subseteq$. The horizontal truth ordering $\leqslant_t$ is, perhaps, less familiar. It can be thought of as a more-true-or-less-false ordering. Again taking Belnap's values as sets of classical values, $\bot$ is less false than *false* because it does not contain *false*, while *false* (as a set) does. Likewise $\top$ is more true than *false*, because it contains *true* while the set *false* does not. In each case, a move to the right corresponds to dropping *false* as a member, or adding *true*.

Both orderings, $\leqslant_t$ and $\leqslant_k$, give us a complete lattice. Let us use $\wedge$ and $\vee$ for meet and join with respect to $\leqslant_t$, and $\otimes$ and $\oplus$ for meet and join with respect to $\leqslant_k$. The notation $\wedge$ and $\vee$ is deliberately suggestive. When restricted to the two truth values *false* and *true*, they are the usual classical connectives, while when restricted to *false*, *true*, and $\bot$ they are the strong Kleene connectives. The $\otimes$ and $\oplus$ operations are less familiar. $\otimes$ is often read as *consensus* and $\oplus$ as *gullability*. They play a role when conflicting classical truth values are involved.

There is a natural notion of negation – a left–right inversion. Set $\neg \mathit{false} = \mathit{true}$, $\neg \mathit{true} = \mathit{false}$, $\neg\top = \top$, and $\neg\bot = \bot$. Again, when restricted to the classical or the Kleene values, we get the corresponding negations of those logics.

Finally, though two orderings have been introduced, they are most decidedly not independent. We have four binary operations: $\wedge$, $\vee$, $\otimes$, and $\oplus$, and thus 12 possible distributive laws. *All of them hold* [19].

Now we can define *four-valued valuations* in the obvious way, as maps from ground atoms to the space of Belnap truth values, again requiring $v(\mathit{false}) = \mathit{false}$ and $v(\mathit{true}) = \mathit{true}$. Both orderings, $\leqslant_t$ and $\leqslant_k$, can be extended to valuations in the usual pointwise fashion. Also the action of four-valued valuations can be extended to all ground formulas: set $v(X \wedge Y)$ to be $v(X) \wedge v(Y)$, and so on. Notice that this even allows us to have $\otimes$ and $\oplus$ in formulas, if desired. Next, a single-step operator can be defined, and it is easiest to base this on the alternate characterizations of $T_{\mathscr{P}}$ and $\Phi_{\mathscr{P}}$ given earlier. We continue to use Φ to denote the operator.

Definition 15. Let $\mathscr{P}$ be a general logic program, and let $\mathscr{P}^{**}$ be as in Definition 5, extended to allow negated atoms in clause bodies. Now, $\Phi_{\mathscr{P}}(v) = w$ where w is the unique valuation determined by the following: if $A \leftarrow B$ is in $\mathscr{P}^{**}$, $w(A) = v(B)$ (where we use Belnap's logic to evaluate $v(B)$).

It is simple to check that $\Phi_{\mathscr{P}}$ is monotone with respect to the $\leqslant_k$ ordering. Now the usual Knaster–Tarski theorem gives us smallest and biggest fixed points. In fact, the smallest fixed point is the same as the smallest fixed point we got when using the Kripke–Kleene semantics. The biggest one is something new of course [12, 13].

There are two orderings available now, not one. If $\mathscr{P}$ has no negations, $\Phi_{\mathscr{P}}$ will also be monotone with respect to the $\leqslant_t$ ordering and so, by Knaster–Tarski again, there will be smallest and biggest fixed points relative to $\leqslant_t$. It is not hard to check that, for $\mathscr{P}$ without negations, the smallest fixed point of $\Phi_{\mathscr{P}}$ with respect to $\leqslant_t$ is classical (that is, the only truth values assigned to any ground atom are *false* and *true*, and in fact, is the same as the smallest fixed point of $T_{\mathscr{P}}$ assigned by the Apt-van Emden–Kowalski semantics. Similarly for the biggest fixed point with respect to $\leqslant_t$.

Earlier we noted an important fact connecting the Apt-van Emden–Kowalski and the Kripke–Kleene semantics: for $\mathscr{P}$ without negations, if the smallest and biggest fixed points of $T_{\mathscr{P}}$ agree on a value for a ground atom A, the smallest fixed point of the three-valued operator $\Phi_{\mathscr{P}}$ also assigns that value; and if the smallest and biggest fixed points of $T_{\mathscr{P}}$ do not agree, A is assigned $\bot$ by the smallest fixed point of $\Phi_{\mathscr{P}}$. Now, if we move to the four-valued setting this fact becomes dramatically nicer to state.

Proposition 16. *Let v_{t} and V_{t} be the smallest and biggest fixed points of the four-valued operator $\Phi_{\mathscr{P}}$ with respect to the $\leqslant_{\mathrm{t}}$ ordering, where $\mathscr{P}$ is a logic program without negations.* (*Recall that these fixed points coincide with the smallest and biggest fixed points of the operator $T_{\mathscr{P}}$.*) *Likewise let v_{k} and V_{k} be the smallest and biggest fixed points of $\Phi_{\mathscr{P}}$ with respect to the $\leqslant_{\mathrm{k}}$ ordering. The result mentioned in the previous paragraph becomes the following simple formula*:

$$v_{\mathrm{k}} = v_{\mathrm{t}} \otimes V_{\mathrm{t}}.$$

But further, *we also have the following items*:

$$V_{\mathrm{k}} = v_{\mathrm{t}} \oplus V_{\mathrm{t}},$$

$$v_{\mathrm{t}} = v_{\mathrm{k}} \wedge V_{\mathrm{k}},$$

$$V_{\mathrm{t}} = v_{\mathrm{k}} \vee V_{\mathrm{k}}.$$

Thus under circumstances where all four fixed points exist, that is, when negations are not involved, all four are closely intertwined. Relationships between the two and the three-valued semantics are really of a general algebraic nature.

Finally, the addition of a fourth truth value does more than simplify the algebra. We can now create a more general notion of logic program, allowing inconsistencies. If an inconsistency arises, that is, if $\top$ is assigned as a truth value, the program can still behave well on parts not involving the inconsistency. By explicitly allowing $\oplus$ and $\otimes$ in program bodies, as well as the eliminable $\vee$ and $\wedge$, we give the programmer considerable freedom to specify what action should be taken in the presence of an inconsistency [12].

7. Stable model semantics

When logic programs without negation are considered, the Apt-van Emden–Kowalski semantics takes *false* as the default, minimal, truth value. Extending semantics to allow negation shifts this to $\perp$ as the default. Is it possible to treat negation, and still take *false* as the default? This is a small part of the motivation behind *stable model semantics*. Stable models arose in the investigation of non-monotonic logic, and were transferred from there to logic programming, where they found a natural home [18, 9]. In this section we sketch the original Gelfond–Lifschitz approach, then in the next we show how this transfers to the four-valued setting.

Let us return to the setting of general logic programs, made up of general program clauses of the form $A \leftarrow L_1, \ldots, L_n$, where A is atomic and $L_1, \ldots, L_n$ are literals. Conjunction and disjunction do not explicitly appear. Now, the idea is to start with a general program $\mathscr{P}$, *and a candidate for a classical Herbrand model* M, and use M to transform the program into a new one without negations, $\mathscr{P}_M$. Since the transformed program has no negations, it has an Apt-van Emden–Kowalski semantics. If that agrees with the model M with which we began, then M is a stable model. We have not yet given the program transformation details, but even without them the following is clear. There is no guarantee that stable models exist, or are unique, and the definition suggests no way of approximating to them. Still, God is in the details, so let us proceed with them.

Definition 17. Let v be a classical, two-valued, valuation, let $\mathscr{P}$ be a general logic program, and let $\mathscr{P}^*$ be as usual. The following is the *Gelfond–Lifschitz* transformation. Modify $\mathscr{P}^*$ as follows:
(i) If $v(A) = true$, remove from $\mathscr{P}^*$ any general clause that has *not A* in its body.
(ii) Next, delete all negative literals from the remaining clause bodies. (If this deletes the entire body of a clause, replace it with *true*.)
Call the resulting set of ground clauses $\mathscr{P}_v$.

The idea behind 1 is obvious: if v is our candidate for a model, and it says we have A, then any clause which requires *not A* for its application is useless and can be removed. After this is done, if *not B* occurs in a clause body, it must be that $v(B) = false$, so if v is our candidate for a model, we can take *not B* for granted, so we may as well delete it and concentrate on the rest of the clause body.

Definition 18. For a general logic program $\mathscr{P}$, $\mathscr{P}_v$ is a set of (*ground*) positive clauses, so $T_{\mathscr{P}_v}$ is monotone, and has a least fixed point. If that least fixed point is v, then v is a *stable model*.

Example 19. Here is the most typical example of stable model semantics. Let $\mathscr{P}$ be the following program:

$$A \leftarrow not\ B,$$

$$B \leftarrow not\ A.$$

Take v to be the valuation such that $v(A) = true$ and $v(B) = false$. Since $v(A) = true$, step 1 of the Gelfond–Lifschitz transformation process causes us to delete the second clause above. Then step 2 removes *not B* from the body of the first clause. We are left with $\mathscr{P}_v$, consisting of the single clause $A \leftarrow true$. In the Apt-van Emden–Kowalski semantics for this, A is *true* and B is *false* – we get v again. Thus v is a stable model.

In a similar way, w is also a stable model, where $w(A) = false$ and $w(B) = true$. These are the only two stable models, and they are incomparable with respect to $\leqslant_t$ – neither is least.

In the example above, the two stable models are both minimal. This always happens: stable models are minimal. Therefore if there are several stable models, they must be incomparable.

For a program having no negations, the unique stable model will also be the least fixed point of its T operator, since the program transformation process changes nothing. Thus stable model semantics fills the requirement of extending the Apt-van Emden–Kowalski semantics, taking *false* as the default, and supplying meanings for at least some programs with negations. Unfortunately, as we noted earlier, stable models need not exist ($A \leftarrow not\ A$ is an example), and if stable models exist there need be no favored one, as we saw above. We return to these issues in the next section.

8. Stable models, generalized

Przymusinski generalized the notion of stable model to allow partiality, or three-valuedness, calling the result *stationary model* semantics [32–34]. We will continue to use the term *stable* here, and when it is necessary to distinguish, we will refer to *two-valued*, *three-valued*, or *four-valued* stable models. Even though Przymusinski presents a three-valued semantics, he keeps the idea of *false* rather than $\perp$ as the default. It solves the problem of programs having no stable model, since one is always guaranteed to exist. It also solves the problem of some programs having many stable models, since there is one that is, in a certain sense, minimal. This minimal stable model was characterized in more than one way – van Gelder et al. [43, 44] gave a construction that led to its standard name, the *well-founded model*. Van Gelder gave an *alternating fixpoint construction* [42]. And Pryzmusinski gave yet another construction that established its minimality [33].

The investigation of stable models fits well with the four-valued approach presented earlier. In addition, extracting the algebraic features behind the constructions makes it clear that they are really quite general. We sketch the four-valued version now, so for the rest of this section the underlying logic is Belnap's, as presented in Section 6. Programs are general logic programs, and we can even allow $\otimes$ and $\oplus$ to appear in program bodies, if desired.

For a general program $\mathscr{P}$ the operator $\Phi_{\mathscr{P}}$ is monotonic with respect to $\leqslant_k$, but it is generally not with respect to $\leqslant_t$. This is the source of the difficulties in taking *false* as default. The key to the solution is to modify the single-step operator so that the role of negation can be isolated. We introduce a two-input single-step operator $\Psi_{\mathscr{P}}(v, w)$ with the idea that input v be used to supply values to positive literals in clause bodies, and input w be used to supply values to negative literals. Apart from this separation of inputs, the output of Ψ is calculated in essentially the same way that the output of Φ was.

Definition 20. Let v and w be two four-valued valuations, mappings from ground atoms to Belnap's four truth values. We define a *pseudo-valuation* $(v \triangle w)$, which is a mapping from ground *literals* to Belnap's space, as follows. For a ground atom A,

$$(v \triangle w)(A) = v(A),$$
$$(v \triangle w)(not\ A) = \neg w(A).$$

The action of a pseudo-valuation is extended to more complicated ground formulas, involving $\wedge$, $\vee$ (and possibly $\otimes$ and $\oplus$) in the expected way, using the various operations of Belnap's logic.

Now, here is the definition of the two-input single-step operator.

Definition 21. Let $\mathscr{P}$ be a general logic program. $\Psi_{\mathscr{P}}(v,w)=u$ where u is the unique valuation determined by the following: if $A \leftarrow B$ is in $\mathscr{P}^{**}$, $u(A)=(v\triangle w)(B)$.

The following items are straightforward to prove, and are the key to what follows.

Proposition 22. *For a general logic program* $\mathscr{P}$:
(i) $\Phi_{\mathscr{P}}(v)=\Psi_{\mathscr{P}}(v,v)$.
(ii) $\Psi_{\mathscr{P}}(v,w)$ *is monotone in both* v *and* w, *with respect to* $\leqslant_{\mathrm{k}}$.
(iii) $\Psi_{\mathscr{P}}(v,w)$ *is monotone in* v, *with respect to* $\leqslant_{\mathrm{t}}$.
(iv) $\Psi_{\mathscr{P}}(v,w)$ *is anti-monotone in* w, *with respect to* $\leqslant_{\mathrm{t}}$ (*that is, if* $w_1\leqslant_{\mathrm{t}}w_2$ *then* $\Psi_{\mathscr{P}}(v,w_1)\geqslant_{\mathrm{t}}\Psi_{\mathscr{P}}(v,w_2)$).

The rest of this section follows from the items in the Proposition above, without any further reference to logic programming details. That is, the rest of this section consists of general facts about operators on Belnap's logic, and these facts are applicable to any programming paradigm meeting the monotonicity/anti-monotonicity conditions of Proposition 22.

Since $\Psi_{\mathscr{P}}$ is monotonic in its first input, with respect to $\leqslant_{\mathrm{t}}$, and four-valued valuations are complete lattices with respect to $\leqslant_{\mathrm{t}}$, the following definition is meaningful.

Definition 23. Let $\mathscr{P}$ be a general logic program. We define a single-input *derived operator*, $\Psi'_{\mathscr{P}}$ as follows:

$$\Psi'_{\mathscr{P}}(w) = \text{the least fixed point, with respect to } \leqslant_{\mathrm{t}}, \text{ of } (\lambda v)\Psi_{\mathscr{P}}(v,w).$$

Now we can give a simple characterization of stable models – but recall, we are in a four-valued setting.

Definition 24. A four-valued *stable model* for general program $\mathscr{P}$ is any fixed point of the derived operator $\Psi'_{\mathscr{P}}$.

This definition relates to earlier work in the following way. Stable models in the Gelfond-Lifschitz sense are the fixed points of $\Psi'_{\mathscr{P}}$ that are two-valued, i.e., that never take on $\bot$ or $\top$ as values. Stationary models in the Przymusinski sense are the fixed points of $\Psi'_{\mathscr{P}}$ that are three-valued, i.e., that never take on $\top$ as a value.

The following are direct lattice-theoretic consequences of the definition and Proposition 22.

Proposition 25. *Let* $\mathscr{P}$ *be a general logic program.*
(i) *If* $\Psi'_{\mathscr{P}}(v)=v$ *then* $\Phi_{\mathscr{P}}(v)=v$.
(ii) $\Psi'_{\mathscr{P}}$ *is monotone with respect to* $\leqslant_{\mathrm{k}}$.
(iii) $\Psi'_{\mathscr{P}}$ *is anti-monotone with respect to* $\leqslant_{\mathrm{t}}$.

The first item above essentially says that stable models must be (supported) models. Since four-valuations constitute a complete lattice with respect to $\leqslant_{\mathrm{k}}$, the second item above guarantees that stable models exist. In particular, there is a smallest one with respect to $\leqslant_k$. It is called the *well-founded model.* The present characterization of it is not the original one of van Gelder et al., but is due to Przymusinski. In general, the well-founded model need not be two-valued. The program $A \leftarrow not\ B$, $B \leftarrow not\ A$, discussed earlier, has two two-valued stable models, but in the well-founded model both A and B evaluate to $\perp$.

The third item above led van Gelder to a most interesting characterization of the well-founded model, using what he called an *alternating fixpoint* approach. It is rather easier to present this approach in our algebraic setting. The main tool is the following, which can be derived from the Knaster–Tarski theorem after observing that if f is anti–monotone, then f^2 is monotone.

Proposition 26. *Let f be anti-monotone on a complete lattice. Then f has a unique pair of extreme oscillation points, a and b. By this we mean the following*:
(i) $a \leqslant b$;
(ii) $f(a)=b$ *and* $f(b)=a$;
(iii) *If $f(x)=y$ and $f(y)=x$ then both x and y are between a and b in the lattice ordering $\leqslant$.*

This proposition is applicable to a derived operator $\Psi'_{\mathscr{P}}$, which must be anti-monotone with respect to $\leqslant_t$ – such an operator must have extreme oscillation points. Now we have the following remarkable fact [15].

Proposition 27. *Let $\mathscr{P}$ be a general logic program, and let v_t and V_t be the extreme oscillation points of $\Psi'_{\mathscr{P}}$, with respect to $\leqslant_t$. Also let v_k and V_k be the smallest and biggest fixed points of $\Psi'_{\mathscr{P}}$, with respect to $\leqslant_k$. (Recall, v_k is the well-founded stable model). Then*

$$v_k = v_t \otimes V_t.$$

So we have a second characterization of the well-founded model as the consensus of extreme oscillation points. But, we also have the following items:

$$V_k = v_t \oplus V_t,$$

$$v_t = v_k \wedge V_k,$$

$$V_t = v_k \vee V_k.$$

This can be summarized in Fig. 1.

9. Bilattices

Belnap's four-valued logic is the simplest of a whole family of similar structures called *distributive bilattices*, due to Matt Ginsberg [19]. All the results in previous

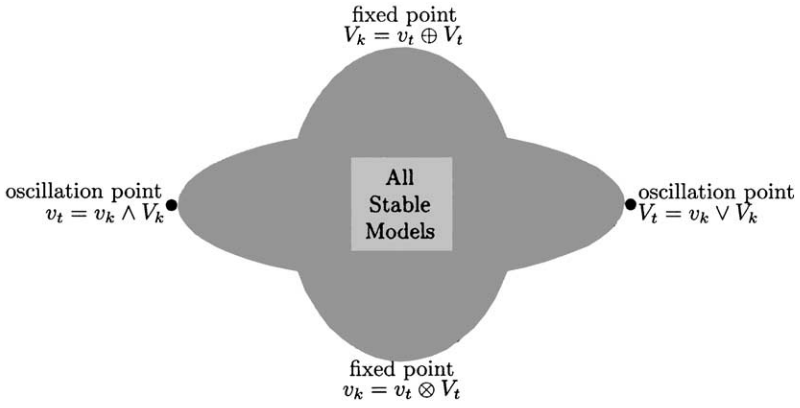


Fig. 1. Distribution of stable models.

sections that made use of Belnap's logic extend to arbitrary distributive bilattices with essentially no changes in proofs. And the more general structures provide a natural setting for useful extensions of the logic programming paradigm.

Definition 28. A *pre-bilattice* is a structure $\mathscr{B}$ with two partial orderings, $\leqslant_t$ and $\leqslant_k$, each making $\mathscr{B}$ a lattice with a top and a bottom. $\mathscr{B}$ is *complete* if each partial ordering makes $\mathscr{B}$ a complete lattice.

In a pre-bilattice, think of $\leqslant_k$ as a knowledge, or information, ordering, and $\leqslant_t$ as a degree of truth ordering, as in Belnap's logic. We continue to use $\otimes$ and $\oplus$ for meet and join with respect to $\leqslant_k$, and $\wedge$ and $\vee$ for meet and join with respect to $\leqslant_t$. We also continue to use $\perp$ and $\top$ for bottom and top with respect to $\leqslant_k$, and *false* and *true* for bottom and top with respect to $\leqslant_t$.

Definition 29. A *distributive bilattice* is a pre-bilattice in which all 12 distributive laws hold. An *infinitely distributive bilattice* is a complete pre-bilattice in which all infinitary, as well as finitary, distributive laws hold.

Belnap's four-valued logic is a distributive bilattice (even an infinitely distributive one, since it is a finite structure). The set of mappings from ground atoms to a distributive (or infinitely distributive) bilattice inherits a distributive (or infinitely distributive) bilattice structure, when pointwise orderings are imposed. We assume this throughout.

There is a standard method of constructing bilattices, due to Ginsberg, that is quite suggestive. Let L_1 and L_2 be two distributive lattices with bottoms and tops. Think of L_1 as a way of measuring evidence *for* propositions, and L_2 as a way of measuring evidence *against*. For instance, in an experimental science, evidence for could be a probability measure, representing degree of confirmation, while evidence against could be, simply, $\{\mathit{false}, \mathit{true}\}$, since one counter-experiment is enough to invalidate a theory. Now, take as the domain of a bilattice the set $L_1 \times L_2$. Thus each member of the bilattice simultaneously encodes evidence for and evidence against. Give $L_1 \times L_2$ the following two orderings:

$$\langle a,b\rangle \leqslant_k \langle x,y\rangle \quad \text{iff } a \leqslant x \text{ and } b \leqslant y,$$

$$\langle a,b\rangle \leqslant_t \langle x,y\rangle \quad \text{iff } a \leqslant x \text{ and } y \leqslant b.$$

(Here $\leqslant$ denotes the ordering of L_1 or L_2, as appropriate.) The $\leqslant_k$ ordering considers an increase as meaning both kinds of evidence have increased. The $\leqslant_t$ ordering considers an increase as meaning the evidence for increased, but the evidence against went down.

If we take $L_1 = L_2 = \{\mathit{false}, \mathit{true}\}$ we get Belnap's four-valued logic (isomorphically, of course). If we take $L_1 = L_2$ to be sets of experts, ordered by inclusion, we get a bilattice capable of dealing with information from more than one source. If we take $L_1 = L_2 = [0, 1]$, we get a 'fuzzy' bilattice – appropriate for the extensions to logic programming proposed in [36, 40]. Many other examples are available. *Ginsberg's Representation Theorem* says that every distributive bilattice is isomorphic to one constructed in the way above. And every infinitely distributive bilattice similarly arises using this construction, starting with lattices satisfying infinitary distributive laws.

In the construction above, if $L_1 = L_2$ an obvious notion of negation is available: $\neg\langle a,b\rangle = \langle b,a\rangle$. That is, we switch around the roles of positive and negative evidence. The representation theorem extends directly to distributive bilattices with negation.

Central fact: Every result stated earlier for logic programming semantics based on Belnap's four-valued logic extends to any infinitely distributive bilattice with negation, with no essential changes in the proof.

As a matter of fact this result can be strengthened to what are called *interlaced bilattices*, in which distributivity conditions are weakened [3]. Also, for many bilattices, natural subsystems can be extracted that are analogous to the classical sublogic or the Kleene three-valued sublogic of Belnap's four-valued logic. (One uses an operation called *conflation* for this – it plays the role for $\leqslant_k$ that negation plays for $\leqslant_t$. We do not give details here, see [16].) These generalizations of the classical or Kleene logics continue to have many of the key properties of the logics they generalize, and are interesting objects of study in themselves.

10. Metric spaces

Consider again the program of Example 5.1, which we repeat here for convenience:

$$\mathit{even}(0) \leftarrow,$$
$$\mathit{even}(s(x)) \leftarrow \mathit{not}\ \mathit{even}(x).$$

This, call it $\mathscr{P}$, is a well-behaved program that still presents some problematic features. We cannot use the Apt-van Emden–Kowalski approach on it, since $T_{\mathscr{P}}$ is not monotone with respect to $\leqslant_t$. Either a three-valued or a four-valued approach provides us with a semantical meaning. It guarantees that $\Phi_{\mathscr{P}}$ has a smallest fixed point with respect to $\leqslant_k$, but this could involve the value $\perp$. It guarantees that the sequence $\Phi_{\mathscr{P}}\uparrow_\alpha$ converges to the smallest fixed point, but this could require as many as Church–Kleene ω_1 steps. As a matter of fact, the smallest fixed point of $\Phi_{\mathscr{P}}$ is two valued – it does not involve $\perp$. Further, it is the only fixed point, hence is the unique stable model as well. And finally, the sequence $\Phi_{\mathscr{P}}\uparrow_\alpha$ converges in ω steps. So the Kripke–Kleene semantics for this program $\mathscr{P}$ happens to have many nice features, but we do not get them from basic theory; we need extra work. In this case, induction arguments are needed to go beyond what the general semantical arguments provide.

This example provides some of the motivation for the introduction of techniques based on metric space methods into the semantics of logic programming. Before giving a general discussion, let us apply metric methods directly to this example. Suppose we define a distance function on the space of valuations. Actually, we can define it for two-valued or three-valued or four-valued valuations – the definition reads the same in any case.

Distance Definition. If v and w are different, set $d(v,w)=1/2^n$, where n is that integer such that $v(even(s^n(0)))\neq w(even(s^n(0)))$, but $v(even(s^k(0)))=w(even(s^k(0)))$ for all $k<n$. And set $d(v,v)=0$.

It is straightforward to check that this definition makes the space of valuations into a complete metric space (whether we consider two-valued, three-valued, or four-valued valuations).

Suppose we consider the space of two-valued valuations for the moment. If valuations v and w agree on $even(s^k(0))$, it is immediate from the single-step operator definition that $T_{\mathscr{P}}(v)$ and $T_{\mathscr{P}}(w)$ will agree on $even(s^{k+1}(0))$. It follows that if the distance between v and w is $1/2^n$, then the distance between $T_{\mathscr{P}}(v)$ and $T_{\mathscr{P}}(w)$ is $1/2^{n+1}$, and thus

$$d(T_{\mathscr{P}}(v),T_{\mathscr{P}}(w))\leqslant \tfrac{1}{2}d(v,w)$$

and so $T_{\mathscr{P}}$ is a *contraction.*

Now by the Banach Contraction Theorem, $T_{\mathscr{P}}$ has a unique fixed point. The same thing applies to $\Phi_{\mathscr{P}}$, hence there is only one fixed point, no matter which semantics is applied. Thus there is a unique stable model. Further, the Banach Contraction Theorem also tells us that we approximate to the unique fixed point in ω steps, independently of starting point.

A space of valuations generally will have many possible metrics on it. A single-step operator will not be a contraction with respect to all of them – the right one must be found. Metrics do not come supplied with a program. In some ways they are like loop invariants in imperative programming, and the suggestion with them is the same as with loop invariants: have a metric in mind when designing a program. That is, have in mind what a single step of the program simplifies.

Logic programmers are not used to thinking in terms of metrics, of course. Here is a notion that is more familiar.

Definition 30. A *level mapping* is a function from ground atoms to natural numbers. We use the notation: $|A|$ is the *level* of ground atom A.

Associated with a level mapping is a metric: set $d(v,w)$ to be $1/2^n$ where valuations v and w differ on some ground atom of level n, but agree on all ground atoms of lower levels. This always gives us a complete metric space. The metric used in the even number example above is simply given by

$$|even(s^n(0))| = n.$$

From now on we define metrics using level mappings, rather than doing so directly.

Example 10.1. Consider the following game program. We have some game in mind, and for it, impossibility of moving constitutes loosing. We assume that each state of the game is encoded by a term t of the language – if the game is chess, for instance, a state consists of the positions of all pieces, the game history, and who is to move next. We also assume there are only a finite number of different possible states.

Now, the program begins with a list of clauses

$$\begin{array}{l} move(a,b) \leftarrow, \\ move(c,d) \leftarrow, \\ \qquad \vdots \end{array}$$

enumerating all possible legal moves of the game. And there is one additional clause:

$$win(x) \leftarrow move(x,y), not\ win(y).$$

If we assume the game has no loops, then the single-step operator for this program has a unique fixed point, and this can be verified as follows.

Define a level mapping by setting $|win(p)|$ to be the height of the game tree with root p – the assumption that the game has no loops gives us finiteness of game trees, so this is well-defined. Arbitrarily set $|move(t,u)|$ to be 1, independently of t and u.

Let d be the metric that corresponds to this level mapping. With respect to this metric, the single-step operator is a contraction – loosely, an application of the single-step operator amounts to making one move in the game, and hence shortening the game tree by one level.

Despite the success of metric methods with these simple programs, various problems remain.

Example 10.2. We elaborate the program of Example 5.1 a little:

$$\begin{aligned}&even(0) \leftarrow,\\&even(s(x)) \leftarrow not\ even(x),\\&any(x) \leftarrow any(x).\end{aligned}$$

Clearly, we want the part involving *even* to behave properly, but the part involving *any* should be allowed to have arbitrary behavior. That is, the single-step operator for this program should have many fixed points, not just one, but they should all agree on *even* and differ on *any*. But once multiple fixed points are involved, the Banach theorem no longer applies, since it guarantees a unique fixed point.

Perhaps the simplest generalization of metrics is to *pseudo-metrics*, which are like metrics except that the distance between objects can be 0 without the objects necessarily being identical. Pseudo-metric spaces carry a natural equivalence relation: call objects equivalent if the distance between them is 0. If a pseudo-metric space is factored using this equivalence relation, the pseudo-metric induces a true metric on the factor space.

Pseudo-metrics give us the additional machinery we need to handle Example 10.2 – in effect we say that valuations can differ on *any* and it does not matter. Technically it is convenient to extend Definition 30 first.

Definition 31. A *partial level mapping* is a function from a subset of ground atoms to natural numbers. We use the notation: $|A|$ is the *level* of ground atom A, if defined.

A partial level mapping induces a pseudo-metric as follows. If v and w agree on all ground atoms A for which $|A|$ is defined, set $d(v,w)=0$. Otherwise, set $d(v,w)=1/2^n$, where v and w differ on some A for which $|A|=n$, but agree on all ground atoms having lower levels. This defines a pseudo-metric, and the factor space defined earlier will be complete, so the Banach theorem can be used after all. This lets us treat Example 10.2 quite directly – we omit details.

Pseudo-metrics, or partial level mappings, also let us deal with programs like the following:

$$\begin{aligned}&even(0) \leftarrow,\\&even(s(x)) \leftarrow not\ even(x),\\&triple(0) \leftarrow,\\&triple(s(s(x))) \leftarrow not\ triple(s(x)), not\ triple(x),\\&sextuple(x) \leftarrow even(x),\ triple(x).\end{aligned}$$

Using partial level mappings, the *even* and the *triple* parts of this can be understood independently by discounting differences on other parts of the program. Then the *sextuple* part can be understood by combining results about *even* and *triple*. More complicated examples are possible. In general, pseudo-metrics seem to be quite a useful tool here.

The application of metric techniques is being extended in several directions. Seda, by using quasi-metric spaces, shows how both the lattice-theoretic and the metric approaches can be combined into a single treatment [35]. Khamsi et al. [21] have shown that more powerful metric fixed-point theorems also have natural applications here.

11. Conclusion

There are two important topics that are common in articles on semantics but that have not yet been mentioned: higher types, and non-determinism. As a matter of fact, both of these have logic programming variants. And both of them are susceptible to the approaches sketched above, though work in these areas is less well-developed.

Existing approaches to higher type logic programming essentially amount to allowing implications to appear within program bodies, thus permitting a kind of modular structure [31]. Such embedded implications have behavioral similarities with intuitionistic implication, and as such have been investigated in [29, 30]. Either an intuitionistic version of negation, or negation as failure can be added. If negation as failure is added, a three-valued approach seems most natural [20].

Example 32. Suppose we have a directed graph, and we wish to write a program that can determine whether there is a path from one node to another. Here is a logic program, using embedded implications and negation as failure, for this purpose. Assume each node has a label, a, b, and so on. The program begins with a list of clauses specifying the graph structure, $edge(a,b) \leftarrow$ for each pair of nodes such that a connecting edge exists. Then, the key item

$$\begin{aligned} path(x,y) \leftarrow\ & edge(x,z), \\ & not\ visited(z), \\ & (visited(z) \Rightarrow path(z,y)). \end{aligned}$$

The idea is, there is a path from x to y if there is an edge from x to z, where z has not yet been visited, and there is a path from z to *y under the assumption that z has been visited.*

Operationally, to establish that $visited(z) \Rightarrow path(z,y)$, the logic engine adds $visited(z)$ to the program, then asks the query $path(z,y)$ of that enlarged program.

This is a simple example of embedded implications. More complex examples could have embedded implications within embedded implications, and so on. But rather nicely, the bilattice approach extends quite naturally to cover such embedded implications, and the entire machinery of stable models carries over as well. The idea is to take as valuations maps from atoms *and programs* to a bilattice. Adding programs as an argument allows relativization to modules. Details of the bilattice approach can be found in [17].

The other item we mentioned was disjunctive logic programming. In this, heads of clauses are allowed to be not just atoms, but disjunctions of atoms. Roughly, such a clause says that if the members of its body are the case, one of the items in its head also is. Semantically, instead of working with valuations, one works with *sets* of valuations, and this suggests that a powerdomain approach should be appropriate. This is, indeed, the case, though the literature so far uses powerdomain theory implicitly, rather than explicitly. On the other hand, the metric approach of Section 10 carries over rather neatly, [21], and shows promise of further extension.

We have tried to show that at least some of the kinds of concerns that are important in developing semantic approaches to imperative and functional programs also arise in logic programming. But logic programming contributes its own twists that make the game a little different, and give the results a strange beauty of their own.

Acknowledgement

I want to thank Howard Blair for a number of suggestions that improved the presentation in this paper.

References

[1] K. Apt, Handbook of Theoretical Computer Science, Vol. B. Elsevier and MIT Press, 1990.
[2] K.R. Apt, M. van Emden, Contributions to the theory of logic programming, J. Assoc. Comput. Mach. 29 (1982) 841–862.
[3] A. Avron, The structure of interlaced bilattices. Math. Struct. Comput. Sci., forthcoming.
[4] N.D. Belnap Jr., A useful four-valued logic, in: J.M. Dunn, G. Epstein (Eds.), Modern Uses of Multiple-Valued Logic, D. Reidel, Dordrecht, 1977.
[5] H.A. Blair, The recursion-theoretic complexity of the semantics of predicate logic as a programming language, Inform. and Control 54 (1/2) (1982) 25–47.
[6] S. Blamey, Handbook of Philosophical Logic, Vol. 3, D. Reidel, Dordrecht, 1985, pp. 1–70 (Chapter-Partial Logic).
[7] K.L. Clark, Negation as failure. In Logic and Databases, H. Gallaire and J. Minker, Eds. Plenum, 1978, pp. 55–76.
[8] K. Doets, From Logic to Logic Programming. MIT Press, Cambridge, MA, 1994.
[9] K. Fine, The justification of negation as failure, in: J.E. Fenstad, I.T. Frolov, R. Hilpinen (Eds.), Logic, Methodology and Philosophy of Science VIII, North-Holland, Amsterdam, 1989, pp. 263–301.
[10] M.C. Fitting, A Kripke/Kleene semantics for logic programs, J. Logic Programming 2 (1985) 295–312.
[11] M.C. Fitting, Partial models and logic programming, Theoret. Comput. Sci. 48 (1987) 229–255.
[12] M.C. Fitting, Bilattices in logic programming, in: G. Epstein (Ed.), The Twentieth Internat. Symp. on Multiple-Valued Logic, IEEE, New York, 1990, pp. 238–246.
[13] M.C. Fitting, Bilattices and the semantics of logic programming, J. Logic Programming 11 (1991) 91–116.
[14] M.C. Fitting, Kleene's logic, generalized, J. Logic Comput. 1 (1992) 797–810.
[15] M.C. Fitting, The family of stable models, J. Logic Programming 17 (1993) 197–225.
[16] M.C. Fitting, Kleene's three-valued logics and their children, Fund. Inform. 20 (1994) 113–131.
[17] M.C. Fitting, On prudent bravery and other abstractions, 1994.
[18] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: R. Kowalski, K. Bowen (Eds.), Proc. of the 5th Logic Programming Symp., MIT Press, Cambridge, MA, 1988, pp. 1070–1080.
[19] M.L. Ginsberg, Multivalued logics: a uniform approach to reasoning in artificial intelligence, Comput. Intelligence 4 (1988) 265–316.

[20] L. Giordano, N. Olivetti, Negation as failure in intuitionistic logic programming, Logic Programming, Proc. Joint Internat. Conf. and Symp. MIT Press, Cambridge, MA, 1992, pp. 430–445.
[21] M.A. Khamsi, V. Kreinovich, D. Misane, A new method of proving the existence of answer sets for disjunctive logic programs: a metric fixed point theorem for multi-valued mappings, J. Logic Programming, forthcoming.
[22] S.C. Kleene, Introduction to Metamathematics, D. Van Nostrand, Princeton, NJ, 1950.
[23] S. Kripke, Outline of a theory of truth, J. Philos. 72 (1975) 690–716. (Reprinted in New Essays on Truth and the Liar Paradox, R. L. Martin (Ed.), Oxford, 1983).
[24] K. Kunen, Answer sets and negation-as-failure, in: J.-L. Lassez (Ed.), Logic Programming, Proc. 4th Internat Conf. MIT Press, Cambridge, MA, 1987, pp. 219–228.
[25] K. Kunen, Negation in logic programming, J. Logic Programming 4 (1987) 289–308.
[26] K. Kunen, Some remarks on the completed database, in: R.A. Kowalski, K.A. Bowen (Eds.), Logic Programming, Proc. 5th Internat. Conf. Symp., The MIT Press, Cambridge, MA, 1988, pp. 978–992.
[27] K. Kunen, Signed data dependencies in logic programming, J. Logic Programming 7 (1989) 231–245.
[28] J.W. Lloyd, Foundations of Logic Programming, 2nd Edition, Springer, Berlin, 1987.
[29] L.T. McCarty, Clausal intuitionistic logic I. fixed point semantics, J. Logic Programming 5 (1988) 1–31.
[30] L.T. McCarty, Clausal intuitionistic logic II, tableau proof procedures, J. Logic Programming 5 (1988) 93–132.
[31] D. Miller, A theory of modules for logic programming, IEEE Symp. on Logic Programming, 1986, pp. 106–114.
[32] T.C. Przymusinski, Stationary semantics for disjunctive logic programs and deductive databases, in: S. Debray, M. Hermenegildo (Eds.), Logic Programming, Proc. 1990 North American Conf., The MIT Press, Cambridge, MA, 1990, pp. 40–59.
[33] T.C. Przymusinski, Well-founded semantics coincides with three-valued stable-semantics, Fund. Inform. 13 (1990) 445–463.
[34] T.C. Przymusinski, Three-valued non-monotonic formalisms and semantics of logic programs, J. Artificial Intelligence (1991).
[35] A.K. Seda, Quasi-metrics and the semantics of logic programs 1996.
[36] E. Shapiro, Logic programs with uncertainties: a tool for implementing rule-based systems, in: A. Bundy (Ed.), Proc. 8th IJCAI, Kaufmann, 1983, pp. 529–532.
[37] R.M. Smullyan, Elementary formal systems (abstract), Bull. Amer. Math. Soc. 62 (1956) 600.
[38] R.M. Smullyan, On definability by recursion (abstract), Bull. Amer. Math. Soc. 62 (1956) 601.
[39] A. Tarski, A lattice-theoretical theorem and its applications, Pacific J. Math. 5 (1955) 285–309.
[40] M. van Emden, Quantitative deduction and its fixpoint theory, J. Logic Programming 3 (1986) 37–53.
[41] M. van Emden, R. Kowalski, The semantics of predicate logic as a programming language, J. Assoc. Comput. Mach. 23 (1976) 733–742.
[42] A. Van Gelder, The alternating fixpoint of logic programs with negation, Proc. 8th ACM Symp. on Principles of Database Systems, ACM, Philadelphia, 1989, pp. 1–10.
[43] A. Van Gelder, K.A. Ross, J.S. Schlipf, Unfounded sets and well-founded semantics for general logic programs, Proc. 7th Symp. on Principles of Database Systems, 1988, pp. 221–230.
[44] A. Van Gelder, K.A. Ross, J.S. Schlipf, The well-founded semantics for general logic programs, J. Appl. Comput. Math. 38 (1991) 620–650.