Negation as Partial Failure

Bamshad Mobasher
Jacek Leszczylowski [1]
Giora Slutzki
Department of Computer Science
Iowa State University
Ames, Iowa 50011
{mobasher, jacek, slutzki}@cs.iastate.edu

Don Pigozzi
Department of Mathematics
Iowa State University
Ames, Iowa 50011
pigozzi@iastate.edu

Abstract

We present a logic programming language which uses a four-valued bilattice as the underlying framework for semantics of programs. The two orderings of the bilattice reflect the concepts of truth and knowledge. The space of truth values includes not only `true` and `false`, but also other truth values which represent no information or conflicting information. Programs are interpreted according to their knowledge content, resulting in a monotonic semantic operator . We present a novel procedural semantics similar to resolution which can retrieve both negative and positive information about a particular goal in a uniform setting. We extend the bilattice-based fixpoint and procedural semantics to incorporate a version of Closed World Assumption. We give soundness and completeness results, with and without the presence of Closed World Assumption. These results are general and are not restricted to ground atomic goals. We further develop the concept of substitution unification and study some of its properties as related to the proposed procedural semantics. Some of these properties may be of independent interest, particularly in the implementation of parallel logic programs.

1 Introduction

Much of the research in the areas of logic programming and deductive databases has attempted to deal with the issue of adequately representing negative or conflicting information. The presence of negation within logic programs, however, causes certain semantic problems [21, 22, 16]. The full inclusion of classical negation in logic programs and queries is generally thought to be infeasible for computational reasons.

Negation as Failure is the most common treatment of negation in logic programming. It is essentially a rule of inference stating that if A is a ground atom, then the goal $\neg A$ succeeds if A fails, and the goal $\neg A$ fails if A succeeds. However, it is well known that Negation as Failure is not sound with respect to classical semantics for programs [21, 22]. There have been many attempts to give a reasonable declarative semantics with respect to which Negation as Failure is sound, including Reiter's Closed World Assumption [20] and Clark's program completion [4]. Unfortunately, while these and other approaches have resulted in various declarative semantics with respect to which Negation as Failure is sound, the corresponding completeness results hold only for restricted sets of logic programs.

[1]On leave from the Institute of Computer Science, Ordona 21, 01-237 Warsaw, Poland.

These semantic problems are also present when the declarative semantics of logic programs involving negation are characterized using fixpoints. Fixpoint semantics for logic programs were originally developed by Van Emden and Kowalski [25] in the context of logic programs without negation. The idea is to associate a natural closure operator T_P on interpretations with each program P and to identify models of P with fixpoints of T_P. The interpretation given by the least fixpoint of T_P is generally taken to be the intended model for the program. When negations are present, however, the T_P operator is generally not monotonic and T_P may have no least fixpoint.

Much of the literature about negation in logic programming examines the ramifications of choosing non-classical semantics based on multi-valued logics [5, 6, 16]. On the other hand, several approaches have proposed dealing with negation by ordering statements and formulas not according to the degree of truth or falsity, but according to the degree of knowledge present in the system about these statements and formulas. Ginsberg [11, 12] introduced a family of multi-valued logics based on certain algebraic structures called *bilattices*, which combined the two aforementioned approaches. Bilattices provided a setting in which one can successfully deal with negation in programs, at least when programs are interpreted according to their knowledge content. It turns out that the fixpoint operator associated with bilattice-based programs will not suffer from the problems discussed above when negation is present in the body of program clauses.

Fitting [7, 8, 9, 10] further studied properties of bilattices. For logic programs based on a certain class of bilattices, he developed a fixpoint semantics and a procedural semantics based on Smullyan style semantic tableaux. The results presented here also use bilattices as the underlying framework for the logic programming language. We use a fixpoint semantics similar to the one proposed by Fitting, but we develop a new procedural semantics based partly on resolution. We use a four-valued logic due to Belnap [2] in which the space of truth values includes not only `true` and `false`, but also other truth values which represent various degrees of knowledge about the truth or falsity of a particular statement, including no information or conflicting information. The space of truth values, and by extension, the space of all interpretations, is now partially ordered in two dimensions using separate orderings. One is called the *truth dimension* and the other is called the *knowledge dimension.* In the truth direction we have all of the machinery of classical logics. In the knowledge dimension, interpreting a program according to its knowledge content gives a monotonic operator associated with that program. When programs are interpreted according to their knowledge content, a statement can potentially be evaluated as both true and false, suggesting the existence of conflicting information. In this sense these programs have the *paraconsistency* property introduced in [3]. To interpret statements according to their knowledge content means that, for instance, a program clause such $A \leftarrow$`true` does not mean that A is true, but rather that there is evidence suggesting that A is true. One advantage of our procedural semantics is that it can retrieve both negative and positive information about a particular goal in a uniform setting.

We also extend the bilattice-based fixpoint and procedural semantics to incorporate a version of Closed World Assumption. This allows inference of negative information when no information is present. We will give soundness and completeness results, with and without the presence of Closed World Assumption. Our soundness and completeness results are general and are not restricted to ground atomic goals.

2 Preliminaries

2.1 Bilattice Background

Bilattices were originally introduced by Ginsberg [11, 12] as the basis for a family of multi-valued logics with certain desirable algebraic properties suitable for combining the notions of truth and knowledge. A bilattice is a space of generalized truth values with two lattice orderings, one measuring degrees of truth, and the other measuring degrees of knowledge. A negation operator provides the connection between the two orderings.

Definition 2.1. A *bilattice* is a structure $[\mathcal{B}, \leq_t, \leq_k, \neg]$ consisting of a non-empty set $\mathcal{B}$, two partial orderings, $\leq_t$ and $\leq_k$, on the elements of $\mathcal{B}$ and a mapping $\neg : \mathcal{B} \rightarrow \mathcal{B}$, such that:

1. each of $\leq_t$ and $\leq_k$ makes $\mathcal{B}$ a complete lattice;

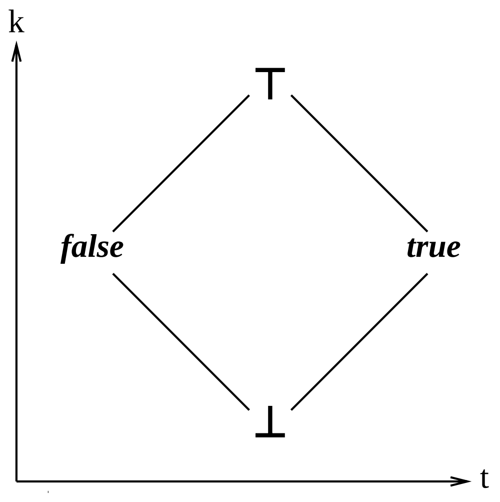

Figure 1: The bilattice $\mathcal{FOUR}$

2. $x \leq_t y$ implies $\neg y \leq_t \neg x$, for all $x, y \in \mathcal{B}$;
3. $x \leq_k y$ implies $\neg x \leq_k \neg y$, for all $x, y \in \mathcal{B}$;
4. $\neg\neg x = x$, for all $x \in \mathcal{B}$.

Informally, we interpret $p \leq_k q$ to mean that the evidence underlying an assignment of the truth value p is subsumed by the evidence underlying an assignment of q. In other words, more is known about the truth or falsity of a statement whose truth value is q than is known about one whose truth value is p. The lattice operations for the $\leq_t$ ordering are natural generalizations of the familiar classical ones. Although negation inverts the degrees of truth as in the case of the classical two-valued logics, in the $\leq_k$ ordering, negation preserves the degree of knowledge about the truth or falsity of a statement.

For the $\leq_t$ ordering, bottom and top elements of the lattice are denoted by `false` and `true`; meet and join are denoted by $\wedge$ and $\vee$; and infinitary meet and join are denoted by $\bigwedge$ and $\bigvee$. For the $\leq_k$ ordering, bottom and top are denoted by $\perp$ and $\top$; meet and join are denoted by $\otimes$ and $\oplus$; and infinitary meet and join are denoted by $\prod$ and $\sum$.

It is easy to see that in a bilattice, `false` and `true` are switched by $\neg$, and that the De Morgan laws hold with respect to $\vee$ and $\wedge$. Furthermore, $\perp$ and $\top$ are left unchanged by $\neg$, and $\oplus$ and $\otimes$ are self-dual under negation. That is, if a and b are elements of the bilattice, then $\neg(a \oplus b) = \neg a \oplus \neg b$ and $\neg(a \otimes b) = \neg a \otimes \neg b$. Distributive bilattices are particularly important in developing semantics for logic programming. There are twelve distributive laws associated with the four operations $\wedge$, $\vee$, $\oplus$,and $\otimes$. A bilattice is *distributive* if all twelve distributivity laws hold. A bilattice meets the *infinite distributivity condition* if all of the infinitary distributive laws such as $a \otimes \bigvee_i b_i = \bigvee_i (a \otimes b_i)$ and $a \wedge \prod_i b_i = \prod_i (a \wedge b_i)$ hold.

Belnap's four-valued logic [2] will serve as the basis and the setting for the logic presented here. The underlying bilattice for this logic, which we call $\mathcal{FOUR}$, consists of the four truth values `true`, `false`, $\perp$, $\top$. The bilattice $\mathcal{FOUR}$ is depicted in Figure 1. In the $\leq_t$ ordering, if $\wedge, \vee, \neg$ are restricted to the two classical truth values `true` and `false`, then they will behave according to the usual two-valued truth table semantics. If they are restricted to the truth values `false`, `true`, and $\perp$, then the behavior is that of Kleene's strong three-valued logic [5, 14]. In the $\leq_k$ ordering, $\otimes$ represents the *consensus* operator which takes the most information consistent with the two arguments. For example `true` $\otimes$ `false` $= \perp$. Similarly, $\oplus$ represents the *accept everything* operator. For instance, `false` $\oplus$ `true` $= \top$.

2.2 Logic Programming Syntax

Our logic programming language, denoted by $\mathcal{L}$, will have the bilattice $\mathcal{FOUR}$ as the underlying space of truth values. The alphabet of $\mathcal{L}$ consists of the usual sets of variables, constants, predicate symbols, and function symbols, similar to conventional logic programming. In addition, it includes the connectives $\leftarrow$, $\neg$, $\wedge$, $\vee$, $\otimes$, and $\oplus$. $\wedge$ and $\vee$ represent the meet and join operations of the bilattice in the truth ordering and $\otimes$ and $\oplus$ represent the meet and join in the knowledge ordering. The quantifiers are $\prod, \sum$ which represent the infinitary meet and join operations of the bilattice in the knowledge ordering.

The notions of *term* and *ground term* are defined in the usual way. The set $U_{\mathcal{L}}$ of all ground terms in a language $\mathcal{L}$ is called the *Herbrand universe* of $\mathcal{L}$ [18]. An *atom* is either one of the constants `true` or `false` or an expression of the form $p(t_1, \cdots, t_n)$, where p is an n-ary predicate symbol and $t_1, \cdots, t_n$ are terms. An atom in which there are no occurrences of variables is called a *ground atom*.

Note that in conventional Prolog, a clause of the form $A \leftarrow$ is taken to stand for $A \leftarrow$ `true`. So the empty clause body is the equivalent of a truth constant. In our language we designate symbols representing each element of the bilattice (with the exception of $\top$ and $\bot$). Hence, the above definition of atomic formulas includes the constants `true` and `false`.

Formulas are either atoms or expressions of the form $\neg A$, $A \oplus B$, $A \otimes B$, $A \wedge B$, or $A \vee B$, where A and B are formulas. A *complex formula* is a formula which is not an atom. A *normalized formula* is a formula in which the operator $\oplus$ does not occur.

A *clause* is an expression of the form:

$$\textstyle\prod_{x_1 \cdots x_n}(A \leftarrow \sum_{y_1 \cdots y_m}(G)),$$

where A is an atom other than `true` and `false`, G is a formula, $x_1, \cdots, x_n$ are variables occurring in A, and $y_1, \cdots, y_m$ are variables occurring in G, but not in A. A is called the *head* and G is called the *body* of the clause. As usual, a *program* is a finite set of clauses. A *goal* is simply a formula. The notions of *normalized clause*, *normalized program*, and *normalized goal* are analogously defined. The *Herbrand Base* of a program P, denoted B_P, is the set of all ground atoms using only constants and function or predicate symbols occurring in P.

Normally, we drop the quantifiers from the clauses and simply write $A \leftarrow G$, where the variables occurring in the head of the clause are implicitly quantified by $\prod$, and the variables occurring in the clause body and not in the clause head are quantified by $\sum$. This convention is a standard practice in logic programming. Of course, in classical logic programming the quantifiers are the truth quantifiers $\forall$ and $\exists$ which are assumed to implicitly quantify a clause. The choice of quantifiers $\prod$ and $\sum$ is motivated by our interest in the knowledge content of statements rather than their truth content.

2.3 Unification and Substitution Unifiers

Before describing the procedural and fixpoint semantics of our logic, we will present some background information on unification, as well as some concepts, dealing with unification of substitutions themselves, which play an important role in our deduction procedure.

Substitutions, renamings, composition of substitutions, unifiers, and most general unifiers are defined in the standard manner as detailed in [18]. The domain of a substitution θ ia denoted by $dom(\theta)$ and the set of variables occurring in the range of θ is denoted by $vrange(\theta)$.

A substitution θ is *idempotent* if $\theta\theta = \theta$. The class of idempotent substitutions exhibit some interesting properties and have been studied extensively [17, 19]. We say that two substitutions θ and σ are *independent* if $dom(\theta) \cap vrange(\sigma) = \emptyset$ and $dom(\sigma) \cap vrange(\theta) = \emptyset$.

It can be shown the mgu's are unique up to renaming of variables. We will sometimes slightly abuse the notation and use this fact to treat $mgu(E_1, E_2)$, where E_1 and E_2 are expressions, as a function returning a unique mgu. In other words, we interpret the equality $\gamma = mgu(E_1, E_2)$ to mean equality up to renaming of variables.

We further extend the notion of unification to substitutions themselves. The notion of unifiable substitutions has been used in concurrent logic programming systems which use AND-parallelism [13]. These substitutions also play an essential role in our procedural semantics.

Definition 2.2. Let σ_1 and σ_2 be substitutions. Then a substitution γ is called a *substitution unifier (s-unifier)* of σ_1 and σ_2, if $\sigma_1\gamma = \sigma_2\gamma$. If such a substitution γ exists, then we say that σ_1 and σ_2 are *unifiable.* γ is a *most general substitution unifier* of σ_1 and σ_2, if for every s-unifier δ of σ_1 and σ_2, there is a substitution

η, such that $\delta = \gamma\eta$. We denote the set of all s-unifiers of σ_1 and σ_2 by $su(\sigma_1, \sigma_2)$ and the set of all most general s-unifiers of σ_1 and σ_2 by $mgsu(\sigma_1, \sigma_2)$.

Most general substitution unifiers are also unique up to renaming of variables. This property is carried over from most general unifiers of two expressions.

Definition 2.3. Let σ_1 and σ_2 be unifiable substitutions. A substitution δ is a *substitution unification* of σ_1 and σ_2, if $\delta = \sigma_1\gamma$ (or equivalently, $\delta = \sigma_2\gamma$), for some $\gamma \in mgsu(\sigma_1, \sigma_2)$. The set of all substitution unifications of σ_1 and σ_2, is denoted by $\sigma_1 \odot \sigma_2$ and is defined by:

$$\sigma_1 \odot \sigma_2 = \{\sigma_1\tau \mid \tau \in mgsu(\sigma_1, \sigma_2)\}.$$

Substitution unifiers are useful in parallel evaluation of queries, since they provide a mechanism for ensuring consistency of the bindings obtained concurrently during the derivation process. Furthermore, they display certain algebraic characteristics which may be of independent interest in unification theory. Here we present some of the properties of substitution unifiers. A more detailed treatment of these and other properties will be presented in the full paper.

Lemma 2.4. *Let θ, η_1, η_2, γ_1, and γ_2 be substitutions. Then:*

1. $su(\eta_1, \eta_2) = su(\gamma_1, \gamma_2)$ *if and only if* $mgsu(\eta_1, \eta_2) = mgsu(\gamma_1, \gamma_2)$.
2. $su(\eta_1, \eta_2) \subseteq su(\theta\eta_1, \theta\eta_2)$.

Under slightly stronger conditions in the previous lemma we can obtain the following interesting right-distributivity result.

Lemma 2.5. *Let θ, η_1, and η_2 be substitutions such that*

$$dom(\eta_i) \subseteq vrange(\theta).$$

Then

$$\theta(\eta_1 \odot \eta_2) = \theta\eta_1 \odot \theta\eta_2.$$

Since most general substitution unifiers of two substitutions σ_1 and σ_2 are unique up to renaming of variables, then so are the substitution unifications of σ_1 and σ_2. In the sequel we interpret $\sigma_1 \odot \sigma_2$ as a function returning a unique substitution unification of σ_1 and σ_2 and interpret the equality $\delta = \sigma_1 \odot \sigma_2$ as equality up to renaming of variables. Using this shorthand notation we can express a weak right-distributivity result in the following manner:

Lemma 2.6. *Let γ, σ_1, and σ_2 be substitutions such that $\sigma_1\gamma$ and $\sigma_2\gamma$ are unifiable. Then σ_1 and σ_2 are unifiable and there is a substitution α such that*

$$\sigma_1\gamma \odot \sigma_2\gamma = (\sigma_1 \odot \sigma_2)\alpha.$$

The idempotent and independent substitutions exhibit some special properties which are useful in the procedural semantics.

Lemma 2.7. *Let η_1 and η_2 be idempotent and independent substitutions. Let $X = dom(\eta_1) \cap dom(\eta_2)$. Then η_1 and η_2 are unifiable if and only if $\eta_1 \mid_X$ and $\eta_2 \mid_X$ are unifiable.*

Finally, we present an important technical lemma which is used in the proof of the Completeness Theorem. Let us denote the variables occuring in an expression E by $vars(E)$.

Lemma 2.8. *Let G_1 and G_2 be two expressions and suppose that θ, σ_1, σ_2, η_1, and η_2 are idempotent and pairwise independent substitutions, and let γ_1 and γ_2 be substitutions, such that the following conditions are satisfied:*

1. *$dom(\theta) = vars(G_1) \cup vars(G_2)$;*
2. *$G_1\theta\eta_1 = G_1\sigma_1\gamma_1$ and $G_2\theta\eta_2 = G_2\sigma_2\gamma_2$;*
3. *η_1 and η_2 are unifiable;*
4. *$dom(\eta_i) = vars(G_i\theta)$, $dom(\sigma_i) = vars(G_i)$, $dom(\gamma_i) = vrange(\sigma_i)$.*

Then, σ_1, σ_2 are unifiable, and there is a substitution γ such that,

$$G_i[\theta(\eta_1 \odot \eta_2)] = G_i[(\sigma_1 \odot \sigma_2)\gamma].$$

3 Logic Programming Semantics

3.1 Fixpoint Semantics

In the classical two-valued logic programming, a single step operator on interpretations, denoted T_P, is associated with a program. In the absence of negation, this operator is monotonic and has a natural least fixpoint. It is this fixpoint which serves as the denotational meaning of the program. However, in the presence of negation in the clause bodies, the T_P operator is no longer monotonic and may not have a fixpoint. The idea of associating such an operator with programs carries over in a natural way to logic programming languages with a distributive bilattice as the space of truth values. However, the ordering in which the least fixpoint is evaluated is the knowledge ordering ($\leq_k$) and not the truth ordering ($\leq_t$). In the $\leq_k$ ordering, negation does not pose any of the problems associated with classical logic programming. The fixpoint semantics presented in this section is essentially due to Fitting [9].

Definition 3.1.

1. An *interpretation* for a program P is a mapping $I : B_P \to \mathcal{FOUR}$.
2. We extend the interpretation I to ground formulas as follows:

$$\begin{array}{rcl} I(\neg A) & = & \neg I(A); \\ I(A_1 \oplus A_2) & = & I(A_1) \oplus I(A_2); \\ I(A_1 \otimes A_2\ ) & = & I(A_1) \otimes I(A_2); \\ I(A_1 \wedge A_2\ ) & = & I(A_1) \wedge I(A_2); \\ I(A_1 \vee A_2\ ) & = & I(A_1) \vee I(A_2). \end{array}$$

3. We further extend the interpretation I to non-ground formulas. For a non-ground formula G:

$$I(G) = \prod\{I(G\sigma) \mid \sigma \text{ is a ground substitution for the variables of } G\}.$$

Pointwise partial orderings are also defined on interpretations in the following manner:

1. $I_1 \leq_k I_2$ if $I_1(A) \leq_k I_2(A)$, for every ground atom $A \in B_P$.
2. $I_1 \leq_t I_2$ if $I_1(A) \leq_t I_2(A)$, for every ground atom $A \in B_P$.

Using this pointwise ordering, the space of interpretations itself becomes a distributive bilattice.

Definition 3.2. The *initial interpretation* I_0 of a program P is defined as follows. For any atom $A \in B_P$:

$$I_0(A) = \begin{cases} \texttt{true} & \text{if } A = \texttt{true} \\ \texttt{false} & \text{if } A = \texttt{false} \\ \bot & \text{otherwise.} \end{cases}$$

Now we can associate a semantic operator with each program.

Definition 3.3. Let P be a program and let $A \in B_P$. The *semantic operator* Φ_P is a function mapping interpretations to interpretations, defined as follows:

$$\Phi_P(I)(A) = \begin{cases} \texttt{true} & \text{if } A = \texttt{true} \\ \texttt{false} & \text{if } A = \texttt{false} \\ \sum\{I(G\sigma) \mid A' \leftarrow G \in P \text{ and } A = A'\sigma\} & \text{otherwise} \end{cases}$$

The Φ_P operator is monotonic with respect to the knowledge ordering. In other words,

$$I_1 \leq_k I_2 \Longrightarrow \Phi_P(I_1) \leq_k \Phi_P(I_2).$$

Now, by the Knaster-Tarski theorem [24], Φ_P has a least fixpoint. It is precisely this least fixpoint which provides the denotational meaning of the program P. In order to approximate the least fixpoint of the operator Φ_P, we use the following notion of upward iteration.

Definition 3.4. The *upward iteration* of Φ_P is defined as follows:

$$\Phi_P \uparrow \alpha = \begin{cases} I_0 & \text{if } \alpha = 0 \\ \Phi_P(\Phi_P \uparrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal} \\ \sum\{\Phi_P \uparrow \beta \mid \beta < \alpha\} & \text{if } \alpha \text{ is a limit ordinal} \end{cases}$$

The smallest ordinal at which this sequence gives the least fixpoint of Φ_P is called the *closure ordinal.* In $\mathcal{FOUR}$ and in fact in any bilattice which satisfies the infinitary distributivity conditions, Φ_P is continuous and its closure ordinal is ω [9].

Using the distributivity and infinitary distributivity properties of the bilattice $\mathcal{FOUR}$ we can establish the semantic equivalence of ordinary and normalized programs.

Theorem 3.5. *Let P be a program over the language $\mathcal{L}$. There exists a normalized program P' over $\mathcal{L}$, such that $\Phi_P = \Phi_{P'}$.*

Since, according to the above theorem, ordinary programs and normalized programs are equivalent, we can now safely concentrate only on normalized programs.

Interpretations over distributive bilattices exhibit some interesting algebraic properties. In particular, we have found the following lemmas useful in the proof of our Soundness Theorem.

Lemma 3.6. *Let G_1 and G_2 be normalized formulas, and suppose I is an interpretation. Then for $\Box \in \{\otimes, \wedge, \vee\}$, we have*

$$I(G_1 \Box G_2) \geq_k I(G_1) \Box I(G_2).$$

Lemma 3.7. *Let θ_1 and θ_2 be unifiable substitutions, let I be an interpretation, and let F be a formula. Then $I(F(\theta_1 \odot \theta_2)) \succeq I(F\theta_i)$, for $i = 1, 2$.*

3.2 Procedural semantics

Fitting's procedural model [8, 9] was based on a version of Smullyan style semantic tableaux [23]. In contrast, we use a resolution-based procedural semantics which will allow us to start with any formula as a goal and within a uniform framework derive both negative and positive information about that goal. In the context of the bilattice $\mathcal{FOUR}$ this means that if the derivation from a goal A leads to success, then A is at least `true`, and if it leads to failure, then A is at least `false`. Informally, if a derivation from A is successful, we say that A has a *proof*, and if the derivation is failed, we say that A has a *refutation.* Note that we do not use the notion of *refutation* in the same way as it is used in resolution-based methods. Also, our notions of successful and failed derivations are quite different from those used in such methods.

Our procedural model is essentially an extension of the well-known operational semantics known as SLDNF-resolution. SLDNF-resolution is based on SLD-resolution [15, 1] augmented with the *negation as* *failure* rule [4]. Negation as Failure uses the notion of *finite failure* to decide if the derivation of a goal has failed. For a definite program P, the *finite failure set* of P, is the set of all ground atoms A for which there exists a finitely failed resolution tree for $P \cup \{\leftarrow A\}$, that is, one which is finite and contains no success branches. A failure branch in such a tree, is one whose leaf node cannot unify with the head of any clause in P.

Our procedural model, called *SLDPF-resolution* (PF stands for Partial Failure) does not require that a finitely failed derivation tree have no success branches. In our approach, the notions of failure and success are treated in exactly the same manner. Thus, a derivation tree, which we call an *SLDPF-tree* for a given goal, can represent both failed and successful derivations (i.e., both refutations and proofs) of that goal. This feature, which we call *Negation as Partial Failure*, is one of the consequences of shifting our emphasis from truth to knowledge. In our derivation trees, each branch of a subtree with the root node A, for some atom A, corresponds to a clause whose head unifies with A. Each such clause is seen as contributing to the information the system has about the truth or falsity of A. All clauses with the same head can be combined using the $\oplus$ operator which, as we explained earlier, is self-dual under negation. In the classical logic programming approach, clauses with the same head are combined using $\vee$, and since the dual of $\vee$ under negation is *wedge*, a failed subgoal is one whose derivation tree has no success branches. In our approach, existence of only one failed branch is sufficient for failure. Thus, we may have goals whose SLDPF-tree has both success and failure branches. Since we interpret free variables in the body of a clause as being quantified by $\sum$, which is its own dual under negation, our procedural semantics will remain sound even in the presence of non-ground negative subgoals.

Note that negative information is derived through the explicit use of clauses of the form $A \leftarrow$ `false`. This approach allows us to treat success and failure in a completely symmetrical manner. Later, we will describe how we can extend Negation as Partial Failure to incorporate the Closed World Assumption with only minor modifications to our procedural and fixpoint semantics.

SLDPF-resolution also extends the treatment of $\neg$ to the operators $\wedge$, $\vee$, and $\otimes$. In other words, if during the derivation a subgoal is reached which is a formula containing one of these operators, then an attempt is made to establish appropriate derivations for the two operands based on the way they act on the elements of the bilattice. This is precisely the point at which we need the notion of substitution unifiers. S-unifiers will ensure that the substitutions obtained from the derivation trees of each operand will not contradict each other once they are finally applied to the formula itself. We now present the details formally in the following definitions.

Definition 3.8. An *SLDPF-tree for* $P \cup \{\leftarrow A\}$, where P is a normalized program and A is an atom, is a (possibly infinite) tree satisfying the following conditions:

1. The root of the tree is A.

2. Let G be a nonleaf node. Then G is an atom and for each clause $G' \leftarrow G'' \in P$, if G and G' are unifiable, then the node has a child $G''\gamma$, where $\gamma = mgu\ (G, G')$. We say that γ is the *substitution associated with the edge* between G and $G''\gamma$.

3. Let G be a leaf node. Then either G is an atom which does not unify with the head of any clause or G is a complex formula.

Definition 3.9. Let E be an expression and let σ be a substitution. The *standardization* of σ with respect to E is a substitution $\theta = (\sigma\ |_{vars(E)}) \cup\ \delta$, where

$$\delta = \{x/y \mid x \in vars(E) - dom(\sigma) \text{ and } y \text{ is a new variable}\}.$$

Definition 3.10. Let P be a normalized program and G a normalized goal. Then

1. G has a *proof of rank 0 with answer* θ if $G =$ `true` and θ is the identity substitution ε. G has a *refutation of rank 0 with answer* θ if $G =$ `false` and θ is the identity substitution ε.

2. G has a *proof of rank* $k+1$ *with answer* θ if:

(a) G is an atom, and $P \cup \{\leftarrow G\}$ has an SLDPF-tree with at least one leaf node G', such that G' has a proof of rank k and with answer θ', and θ is the standardization of $\sigma_1 \cdots \sigma_n \theta'$ with respect to G, where $\sigma_1, \cdots, \sigma_n$ are the substitutions associated with each edge along the path from G to G'; or

(b) G is $\neg G'$, and G' has a refutation of rank k with answer θ; or

(c) G is $G_1 \otimes G_2$ or $G_1 \wedge G_2$, G_1 and G_2 have proofs of ranks k_1 and k_2 with answers θ_1 and θ_2, respectively, $k = max(k_1, k_2)$, and θ is the standardization of $\theta_1 \odot \theta_2$ with respect to G; or

(d) G is $G_1 \vee G_2$, G_1 or G_2 has a proof of rank k with answer θ', and θ is the standardization of θ' with respect to G.

3. G has a *refutation of rank* $k+1$ *with answer* θ if:

(a) G is an atom, and $P \cup \{\leftarrow G\}$ has an SLDPF-tree with at least one leaf node G' such that G' has a refutation of rank k and with answer θ', and θ is the standardization of $\sigma_1 \cdots \sigma_n \theta'$ with respect to G, where $\sigma_1, \cdots, \sigma_n$ are the substitutions associated with each edge along the path from G to G'; or

(b) G is $\neg G'$, and G' has a proof of rank k, with answer θ; or

(c) G is $G_1 \otimes G_2$ or $G_1 \vee G_2$, G_1 and G_2 have refutations of ranks k_1 and k_2 with answers θ_1 and θ_2, respectively, $k = max(k_1, k_2)$, and θ is the standardization of $\theta_1 \odot \theta_2$ with respect to G; or

(d) G is $G_1 \wedge G_2$, G_1 or G_2 has a refutation of rank k with answer θ' and θ is the standardization of θ' with respect to G.

Definition 3.11. Let P be a normalized program and G a normalized goal. Then G has a proof (respectively, a refutation) with answer θ, if G has a proof (respectively, a refutation) of rank k, with answer θ, for some $k \geq 0$.

We also adopt the standard process of using suitable variants of program clauses at each step of a proof or refutation. This is so that the variables used for the derivation do not already occur in the derivation up to that point.

3.3 Basic Results

In this section we present the soundness and completeness results for Negation as Partial Failure. These theorems establish the correspondence between the procedural and the fixpoint semantics. For the purpose of these theorems we will denote the ordering in the knowledge lattice by $\preceq$.

Theorem 3.12 (Soundness). *Let P be a normalized program, G a normalized goal, and θ a substitution for the variables of G.*
If G has a proof with answer θ, then $(\Phi_P \uparrow \omega)(G\theta) \succeq$ `true`;
If G has a refutation with answer θ, then $(\Phi_P \uparrow \omega)(G\theta) \succeq$ `false`.

The key to the proof of the Completeness Theorem is the following lifting lemma. It generalizes the lifting lemma which is used in establishing the completeness of SLD-resolution (see [18]).

Lemma 3.13 (Lifting Lemma). *Suppose that $G\theta$ has a proof (respectively, refutation) with answer η. Then G has a proof (respectively, refutation) with answer σ, such that $G\theta\eta = G\sigma\gamma$, for some substitution γ.*

Proof: (By induction on k)
Basis: ($k = 0$)
Suppose that $G\theta$ has a proof of rank 0 with answer η. Then $G\theta$ must be the constant `true`. Hence, $G =$ `true` and $\eta = \varepsilon$. But $G =$ `true` has a proof of rank 0 with answer ε. Clearly, $\theta\eta = \theta\varepsilon = \theta = \varepsilon\theta$. Now, take $\sigma = \varepsilon$ and let $\gamma = \theta$. By a similar argument, if $G\theta$ has a refutation of rank 0, then G has a refutation of rank 0 with answer σ, such that $\sigma = \varepsilon$ and $\gamma = \theta$.

Induction: Assume the result holds for proofs and refutations of rank k. We present the argument for proofs of rank $k+1$; the argument for refutations is similar. Suppose that $G\theta$ has a proof of rank $k+1$ with answer η.

When G is an atom A, then $P \cup \{\leftarrow A\theta\}$ has an SLDPF-tree with at least one success branch. Furthermore, the corresponding leafnode F has a proof of rank k with answer ρ, such that η is the standardization of $\sigma_1 \cdots \sigma_n \rho$ w.r.t. $A\theta$, where $\sigma_1, \cdots, \sigma_n$ are substitutions associated with the edges along the path in the tree from $A\theta$ to F. The result is proved by a secondary induction on the length n of this path similar to the proof of the lifting lemma for SLD-resolution. If G is $G_1 \otimes G_2$ then $G_1\theta$ has a proof of rank k_1 with answer η_1 and $G_2\theta$ has a proof of rank k_2 with answer η_2, $k = max(k_1, k_2)$, and η is the standardization of $\eta_1 \odot \eta_2$ with respect to $G\theta$. Now, by the inductive hypothesis, G_1 has a proof of rank k_1 with answer σ_1, such that $G\theta\eta_1 = G\sigma_1\gamma_1$, for some substitution γ_1, and G_2 has a proof of rank k_2 with answer σ_2, such that $G\theta\eta_2 = G\sigma_2\gamma_2$, for some substitution γ_2. Hence, $G_1 \otimes G_2$ has a proof of rank $max(k_1, k_2) + 1 = k + 1$. It is easy to verify that the conditions of lemma 2.8 are satisfied. Hence, $\sigma_1 \odot \sigma_2$ exists, and furthermore, there is a substitution γ' such that $G_i[\theta(\eta_1 \odot \eta_2)] = G_i[(\sigma_1 \odot \sigma_2)\gamma']$. Now, $G_i\theta\eta = G_i\sigma\gamma$, where σ is an appropriate standardization of $\sigma_1 \odot \sigma_2$ w.r.t. $G_1 \otimes G_2$ and γ is the appropriate standardization of γ' w.r.t. $(G_1 \otimes G_2)\sigma$. Hence, $G_1 \otimes G_2$ has proof of rank $k+1$ with answer σ and

$$(G_1 \otimes G_2)\theta\eta = (G_1 \otimes G_2)\sigma\gamma.$$

The result is proved in a similar manner when $G = G_1 \wedge G_2$, $G = G_1 \vee G_2$, and $G = \neg G'$. ∎

Theorem 3.14 (Completeness). *Let P be a normalized program and G a normalized goal. Suppose θ is a substitution for the variables of G.*
If $(\Phi_P \uparrow \omega)\ (G\theta) \succeq$ `true`, then G has a proof with answer σ, such that $G\theta = G\sigma\gamma$, for some substitution γ;
If $(\Phi_P \uparrow \omega)\ (G\theta) \succeq$ `false`, then G has a refutation with answer σ such that $G\theta = G\sigma\gamma$, for some substitution γ.

4 Incorporating Closed World Assumption

In this section we will present a modified version of Negation as Partial Failure which incorporates a version of the Closed World Assumption. In standard logic programming the Closed World Assumption (CWA) will allow one to deduce negative information. CWA is essentially an inference rule stating that if a ground atom A is not a logical consequence of a program, then infer $\neg A$. This inference rule, introduced by Reiter [20], is often a natural rule to use when dealing with databases.

We incorporate the Closed World Assumption into our logic by implicitly adding to the program clauses of the form $A \leftarrow$ `false`, for each atom which does not unify with the head of any clause in the original program. In this way, any subgoal which does not unify with the head of any clause in the original program will have a refutation with respect to the extended program.

In the remainder of this section, we will describe the changes that need to be made to our procedural and fixpoint semantics, and we will establish the soundness and completeness results of Negation as Partial Failure with the Closed World Assumption. We first define the notions of *closed world refutation* and *closed world proof.* These definitions are for the most part identical to the notions of proof and refutation as defined earlier.

Definition 4.1. Let P be a normalized program and G a normalized goal. Then G has a *closed world proof (cw-proof) of rank 0 with answer* θ with respect to P if $G =$ `true` and θ is the identity substitution ε. G has a *closed world refutation (cw-refutation) of rank 0 with answer* θ with respect to P if $G\theta$ is an atom, other than `true`, which does not unify with the head of any clause in P. The notions of *cw-proofs* and *cw-refutations of rank* $k+1$ are defined in a similar manner as those of proofs and refutations in the previous section.

The most significant departure from the earlier notions of proof and refutation is in the way that a cw-refutation of rank 0 is established. We now have a failed derivation (a refutation), not only if we end up with a subgoal `false`, but when we are left with any subgoal, including `false`, which does not unify with the head of any clause in the normalized program. Here the resemblance to the traditional notion of Negation as Failure should be clear.

Next we describe the new fixpoint semantics which will incorporate the Closed World Assumption. Here also the primary difference with our original fixpoint semantics is in the definition of the initial interpretation.

Definition 4.2. The *initial interpretation* I_0^{cw} of a program P is defined as follows. For any atom $A \in B_P$:

$$I_0^{cw}(A) = \begin{cases} \texttt{true} & \text{if } A = \texttt{true} \\ \texttt{false} & \text{if } A \neq \texttt{true} \text{ and } A \text{ does not unify with the head of} \\ & \text{any clause in } P \\ \bot & \text{otherwise} \end{cases}$$

Now, with each program P, we associate a new semantic operator denoted by Φ_P^{cw} which is defined as follows.

Definition 4.3. Let P be a program and let $A \in B_P$. The *closed world semantic operator* Φ_P^{cw} is a function mapping interpretations to interpretations, defined as follows:

$$\Phi_P^{cw}(I)(A) = \begin{cases} \texttt{true} & \text{if } A = \texttt{true} \\ \texttt{false} & \text{if } A \neq \texttt{true} \text{ and } A \text{ does} \\ & \text{not unify with the} \\ & \text{head of any clause in } P \\ \sum\{I(G\sigma) \mid A' \leftarrow G \in P\; A = A'\sigma\} & \text{otherwise} \end{cases}$$

Definition 4.4. The *upward iteration* of Φ_P^{cw} is defined as follows:

$$\Phi_P^{cw} \uparrow \alpha = \begin{cases} I_0^{cw} & \text{if } \alpha = 0 \\ \Phi_P^{cw}(\Phi_P^{cw} \uparrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal} \\ \sum\{\Phi_P^{cw} \uparrow \beta \mid \beta < \alpha\} & \text{if } \alpha \text{ is a limit ordinal} \end{cases}$$

We shall see below that the closed world procedural and fixpoint semantics can be reduced to the corresponding semantics without the Closed World Assumption. For this purpose we must extend the notions of proof and refutation as well as those of the semantic operator and upward iteration (as defined originally for normalized programs) to apply to infinite normalized programs.

We now define a special class of programs which provide the technical tool for specifying the relationship between our original semantics and the closed world semantics of normalized programs.

Definition 4.5. Let P be a normalized program. Then the *normalized extension* of P, denoted P^+, is a possibly infinite program defined as follows.

1. For every clause $C \in P$, $C \in P^+$.
2. For every atom $A \notin \{\texttt{true},\texttt{false}\}$ which does not unify with the head of any clause in P, the clause $A \leftarrow \texttt{false} \in P^+$.

The next lemma will establish the relationship between the fixpoint semantics for extended normalized programs and the closed world fixpoint semantics for normalized programs.

Lemma 4.6. *Let P be a normalized program and G a normalized goal. Then:*

$$(\Phi_P^{cw} \uparrow n)(G) = (\Phi_{P+} \uparrow n)(G)$$

for every $n < \omega$.

Clearly, the above lemma implies that:

$$(\Phi_P^{cw} \uparrow \omega)(G) = (\Phi_{P+} \uparrow \omega)(G).$$

Similarly, the following two lemmas will specify the relationship between the procedural semantics for extended normalized programs and the closed world procedural semantics for normalized programs.

Lemma 4.7. *Let P be a normalized program and G a normalized goal. If G has a cw-proof (respectively, a cw-refutation) of rank $k \geq 0$ with answer σ, with respect to P, then G has a proof (respectively, a refutation) of rank k or $k+1$ with answer σ, with respect to P^+.*

Lemma 4.8. *Let P be a normalized program and G a normalized goal. If G has a proof (respectively, a refutation) of rank $k \geq 0$ with answer σ, with respect to P^+, then G has a cw-proof (respectively, a cw-refutation) of rank k or $k-1$ with answer σ, with respect to P.*

Now, we have all we need in order to prove the soundness and completeness theorems for the Negation as Partial Failure with the Closed World Assumption.

Theorem 4.9 (Closed World Soundness). *Let P be a normalized program, G a normalized goal, and θ a substitution for the variables of G. If G has a cw-proof with answer θ with respect to P, then $(\Phi_P^{cw} \uparrow \omega)(G\theta) \succeq$ `true`, and if G has a cw-refutation with answer θ with respect to P, then $(\Phi_P^{cw} \uparrow \omega)(G\theta) \succeq$ `false`.*

Proof: Suppose that G has a cw-proof (respectively, a cw-refutation) of rank k with answer θ with respect to P, for some $k \geq 0$. Then by lemma 4.7, G has a proof (respectively, a refutation) of rank k or $k+1$ with answer θ, with respect to P^+. Then by the soundness theorem for normalized programs (now applied to possibly infinite programs) we can conclude that $(\Phi_{P+} \uparrow \omega)(G\theta) \succeq$ `true` (respectively, `false`). Finally, by lemma 4.6, we have $(\Phi_P^{cw} \uparrow \omega)(G\theta) \succeq$ `true` (respectively, `false`). ■

Theorem 4.10 (Closed World Completeness). *Let P be a normalized program and G a normalized goal. Suppose θ is a substitution for the variables of G. If $(\Phi_P^{cw} \uparrow \omega(G\theta) \succeq$ `true`, then G has a cw-proof with respect to P and with answer σ, such that $G\theta = G\sigma\gamma$, for some substitution γ; and if $(\Phi_P^{cw} \uparrow \omega(G\theta) \succeq$ `false`, then G has a cw-refutation with respect to P and with answer σ, such that $G\theta = G\sigma\gamma$, for some substitution γ.*

Proof: Suppose that $(\Phi_P^{cw} \uparrow \omega(G\theta) \succeq$ `true` (respectively, `false`). Then by lemma 4.6 we have $(\Phi_{P+} \uparrow \omega)(G\theta) \succeq$ `true` (respectively, `false`). Then by the completeness theorem for normalized programs (now applied to possibly infinite normalized programs) G has a proof (respectively, a refutation) of rank k, for some $k' \geq 0$, with answer σ, with respect to P^+, such that $G\theta = G\sigma\gamma$, for some substitution γ. Now, by lemma 4.8, we conclude that G has a cw-proof (respectively, a cw-refutation) of rank k, with answer σ, with respect to P, such that $G\theta = G\sigma\gamma$, for some substitution γ, and $k = k'$ or $k = k' - 1$. ■

5 Work in Progress

There are two areas in which the work presented in this paper is continuing. One is motivated by the asymmetry of the notions of success and failure when the Closed World Assumption is incorporated into the procedural semantics. In the presence of the Closed World Assumption, in order to obtain answers for refutations, the system must return a substitution encoding the fact that the goal does not unify with the head of *any* clause. This can be done using the notion of *nonunification*. However, since in general there may be infinitely many most general nonunifiers of two expressions, we use the novel notion of *scheme substitutions* in order to represent these nonunifiers in a finite manner. Nonunifiers and scheme substitutions may also be of independent interest in other areas of logic programming.

The second area involves the extension of the procedural semantics, presented here for the bilattice $\mathcal{FOUR}$, to general distributive bilattices. Such an extension would require the classification of the bilattice elements which are join-irreducible in the knowledge ordering. Furthermore, the study of these elements may shed more light on certain algebraic properties of multi-valued logics which are based on bilattices.

The results of research in these areas will be presented separately in forthcoming papers.

References

[1] K. R. Apt and M. H. van Emden, Contribution to the theory of logic programming, *JACM*, **29** (1982), pp. 841-862.

[2] N. D. Belnap, Jr. A usefull four-valued logic, in *Modern Uses of Multiple-Valued Logic*, J. Michael Dunn and G. Epstein editors, D. Reidel, Boston (1977), pp. 8-37.

[3] H. A. Blair and V. S. Subrahmanian, Paraconsistent foundations for logic programming, *Journal of Non-Classical Logic*, **5** (1982), pp. 45-73.

[4] K. L. Clark, Negation as failure, in *Logic and Data Bases*, H. Gallaire and J. Minker editors, Plenum Press, New York (1978), pp. 293-322.

[5] M. C. Fitting, A Kripke/Kleene semantics for logic programs, *Journal of Logic Programming*, **4** (1985), pp. 295-312.

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

[7] M. C. Fitting, Logic programming on a topological bilattice, *Fund. Informatica*, **11** (1988), pp. 209-218.

[8] M. C. Fitting, Negation as refutation, in *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, R. Parikh editor, IEEE (1978), pp. 63-70.

[9] M. C. Fitting, Bilattices in logic programming, in *The Twentieth International Symposium on Multiple-Valued Logic*, G. Epstein editor, IEEE (1990), pp. 63-70.

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

[11] M. L. Ginsberg, Multi-valued logics, in *Proceedings of the Fifth National Conference on Artificial Intelligence, AAAI-86*, Morgan Kaufmann, Los Altos, CA (1986), pp. 243-247.

[12] M. L. Ginsberg, Multi-valued logics: a uniform approach to reasoning in artificial intelligence, *Computational Intelligence*, **4** (1988), pp. 265-316.

[13] J. M. Jacquet, *Conclog: A Methodological Approach to Concurrent Logic Programming*, Springer-Veralg, Berlin (1991).

[14] S. C. Kleene, *Introduction to Metamathematics*, Van Nostrand Reinhold (1957).

[15] R. A. Kowalski, Predicate logic as a programming language, in *Information Processing 74*, Stockholm, North Holland (1974), pp. 569-774.

[16] K. Kunen, Negation in logic programming, *Journal of Logic Programming*, **4** (1987), pp. 289-308.

[17] J. L. Lassez and M. J. Maher and K. Marriott, Unification revisited, in *Foundations of Deductive Databases and Logic Programming*, J. Minker editor, Morgan Kaufmann, Los Altos, CA (1988), pp. 587-625.

[18] J. W. Lloyd, *Foundations of Logic Programming*, second edition, Springer, Berlin (1987).

[19] C. Palamidessi, Algebraic properties of idempotent substitutions, in *Proccedings of the 17th International Colloquium on Automata, Languages and Programming*, M. S. Paterson editor, Springer-Verlag, Berlin (1990), pp. 386-399.

[20] R. Reiter, On closed world data bases, in *Logic and Data Bases*, H. Gallaire and J. Minker editors, Plenum Press, New York (1978), pp. 55-76.

[21] J. C. Shepherdson, Negation in logic programming, in *Foundations of Deductive Databases and Logic Programming*, J. Minker editor, Morgan Kaufmann, Los Altos, CA (1988), pp. 19-88.

[22] J. C. Shepherdson, Logics for negation as failure, in *Logic from Computer Science*, Y. N. Moschovakis editor, Springer-Verlag, Berlin (1990), pp. 521-583.

[23] R. M. Smullyan, *First Order Logic*, Springer-Verlag, Berlin (1968).

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

[25] M. van Emden and R. A. Kowalski, The semantics of predicate logic as a programming language, *JACM*, **23** (1976), pp. 733-742.