Semantic Issues in Deductive Databases and Logic Programs

Halina Przymusinska
Department of Computer Science
The University of Texas at El Paso
(fp00@utep.bitnet)

Teodor Przymusinski
Department of Mathematical Sciences
The University of Texas at El Paso
(utep-vaxa!teodor@cs.utexas.edu)

To our sons Lukasz and Marcel with love

1 Introduction

The theory of *deductive databases* traces its beginnings to the fundamental paper written by Codd [Cod70] in which formal foundations of the so called *relational databases* have been first outlined. A relational database is a collection of *individual facts (data)*, equipped with the capability to efficiently manipulate (update) its contents and to answer queries about it. Typically, *relational algebra* is used to implement those functions.

In the middle of 1970's it was realized, however, that, in spite of their great usefulness, the capabilities of relational databases are severely limited by their inability to handle *deductive* and *incomplete* information. The need for *deductive reasoning* stems from the fact that we often want to be able to deduce *new* information from the *facts* already present in the database and from *deductive rules*, which are known to us and which we could include in the database. Unfortunatelly, relational databases do not have the capability to store and handle sufficiently general deductive rules. The necessity to deal with *incomplete information* is due to the fact that our knowledge is often incomplete and therefore we are forced to derive conclusions in the absence of complete information. This phenomenon is particularly evident in the case of *disjuctive* and *negative* information. Again, relational databases do not provide satisfactory capabilities to deal with this problem.

The above realization led to the concept of *deductive databases*, which, in addition to storing individual facts (extensional data), can store and manipulate deductive rules of reasoning (intensional data) and are able to answer queries based on logical derivation coupled with some mechanism for handling incomplete information. The initial impetus to the new field has been given by the first Workshop on Logic and Databases held in Toulouse in 1977 [GM78].

Several years later, Reiter made a first attempt at providing a theoretical foundation for deductive databases [Rei84] and the first survey of the field appeared [GMN84]. The area has been developing rapidly during the following years. In particular, the first comprehensive book on the subject [Ull88] has been recently published and implementation of large experimental deductive database systems [MUG86, Zan88] has begun. More information on the historical perspective of the field can be found in [Min88b].

Logic programming was introduced in the early 1970's by Colmerauer [CKRP73] and Kowalski [Kow74] and the first Prolog interpreter was implemented by Roussel in 1972 [Rou75]. Logic programming introduced to computer science the important concept of *declarative* – as opposed to *procedural* – programming. It is based on Kowalski's principle of *separation of logic and control* [Kow74, Kow79]. Ideally, a programmer should be only concerned with the *declarative meaning* of his program, while the procedural aspects of program's execution are handled automatically.

Late 1970's witnessed the beginning of the development of formal foundations of logic programming, starting with the classical papers by Van Emden and Kowalski on the least model semantics [VEK76], Clark's work on program completion [Cla78] and Reiter's paper on the closed world assumption [Rei78]. Further progress in this direction was achieved in the early 1980's (e.g., [**?**, JLL83]) leading to the appearance of the first book devoted to foundations of logic programming [Llo84, Llo87]. The selection of Prolog as the underlying language of the Japanese Fifth Generation Computers Project in 1982 led to a much greater visibility of logic programming and rapid proliferation of various logic programming languages – especially Prolog.

It has quickly become clear that logic programming and deductive databases are closely related [Rei84, GMN84, LT85, LT86, Min88a] and that they are based on very similar theoretical foundations. Their introduction and subsequent development of their formal foundations has been an outgrowth and an unquestionable success of the *logical approach* to *knowledge representation*. This approach is based on the idea of providing intelligent machines with a *logical specification* of the knowledge that they possess, thus making it independent of any particular implementation, context-free, and easy to manipulate, exchange and reason about.

Consequently, a precise *meaning* or *semantics* must be associated with any logic or database program in order to provide its declarative specification. The performance of any *computational mechanism* is then evaluated by comparing its behavior to the *specification* provided by the declarative semantics. This does not mean that such a computational mechanism must necessarily be based on some logical proof procedure, such as, for example, resolution. It implies however, that *logic is the final arbiter of its correctness* [Rei84].

Finding a suitable declarative or intended semantics of deductive databases and logic programs is one of the most important and difficult research problems and is the main topic of this paper. In particular, the paper reports on a very

significant progress made recently in this area. It also presents some results which have not yet appeared in print.

The paper is organized as follows. In the next two sections we define deductive databases and logic programs. Subsequently, in Sections 4 and 5, we discuss model theory and fixed points, which play a crucial role in the definition of semantics. Section 6 is the main section of the paper and is entirely devoted to a systematic exposition and comparison of various proposed semantics. In Section 7 we discuss the relationship between declarative semantics of deductive databases and logic programs and non-monotonic reasoning. Section 8 contains concluding remarks.

2 Deductive Databases

By a *deductive database* $\overline{P}$ we mean a triple:

$$\overline{P} = < P, SEM(P), IC >$$

where P is a *logic program* (also called a *database program*), SEM(P) is the *declarative semantics* of P and IC represents the *integrity constraints*.

Logic (or database) programs P consist of finitely many clauses and are defined in the next section. Semantics SEM(P) of a program P is the intended meaning of P and usually is defined either as a completion COMP(P) of the program P or as an intended model M_P or a set of intended models $MOD(P)$ of P. The problem of finding a suitable semantics $SEM(P)$ for a program P and a discussion of some of the proposed solutions is the main topic of this paper and is investigated in great detail in the sequel. However, any semantics is assumed to logically imply all formulae from P:

$$SEM(P) \models P.$$

Integrity constraints usually consist of a finite set of (closed) first order formulae and are supposed to be at all times satisfied by the semantics SEM(P) of the program P:

$$SEM(P) \models IC.$$

Although there is no way to precisely differentiate between formulae which should be considered as integrity constraints and those which should be made part of the program, nevertheless the overall distinction between the respective roles of the program P and the integrity constraints IC is quite essential:

- Integrity constraints do not directly affect the semantics SEM(P) of the database $\overline{P}$, which is entirely determined by the program P. Instead, they are used only to verify the integrity of the program P. Namely, any program P whose semantics SEM(P) does not satisfy the integrity constraints IC is considered to violate the imposed constraints and is therefore rejected.

This is in contrast to the fact that updates (changes) to the program P itself generally lead to immediate changes of the semantics SEM(P).

- Once the integrity of the program P has been established, integrity constraints are no longer needed for query answering, which depends entirely on SEM(P).
- Integrity constraints describe properties and relations which are supposed to be satisfied at all times by the program and therefore they remain largely static. On the other hand, the database program P usually changes frequently due to various updates (deletions, insertions etc.).
- Integrity constraints may, in general, include arbitrary first order formulae, whereas programs usually consist of sets of clauses.

For example, if P is any logic program, then we could define the semantics SEM(P) of P as the Clark completion $comp(P)$ of the program [Cla78] and the integrity constraints IC to consist of the so called Clark Equality Axioms (see [Llo84, Kun87, Prz89c]).

3 Logic Programs

By an *alphabet* $\mathcal{A}$ of a first order language $\mathcal{L}$ we mean a (finite or countably infinite) set of *constant*, *predicate* and *function* symbols[1]. In addition, any alphabet is assumed to contain a countably infinite set of *variable* symbols, connectives $(\wedge, \vee, \neg, \leftarrow)$, quantifiers $(\exists, \forall)$ and the usual punctuation symbols. A *term* over $\mathcal{A}$ is defined recursively as either a variable or a constant or an expression of the form $f(t_1, \ldots, t_k)$, where f is a function symbol and t_i's are terms. An *atom* over $\mathcal{A}$ is an expression of the form $p(t_1, \ldots, t_k)$, where p is a predicate symbol and t_i's are terms. The *first order language* $\mathcal{L}$ over the alphabet $\mathcal{A}$ is defined as the set of all well-formed first order formulae that can be built starting from the atoms and using connectives, quantifiers and punctuation symbols in a standard way. A *literal* is an atom or its negation. An atom A is called a *positive literal* and its negation $\neg A$ is called a *negative literal*. An expression is called *ground* if it does not contain any variables. The set of all ground atoms of $\mathcal{A}$ is called the *Herbrand base* $\mathcal{H}$ of $\mathcal{A}$. If G is a quantifier-free formula, then by its *ground instance* we mean any ground formula obtained from G by substituting ground terms for all variables. For a given formula G of $\mathcal{L}$ its *universal closure* or just *closure* $(\forall)G$ is obtained by universally quantifying all variables in G which are not bound by any quantifier. Unless otherwise stated, all formulae are assumed to be *closed*.

[1] The set of function symbols may be empty while the sets of constant and predicate symbols are assumed to be non-empty.

By a *logic program* we mean a finite set of universally closed *clauses* of the form

$$A \leftarrow L_1 \wedge \ldots \wedge L_m$$

where $m \geq 0$, A is an atom and L_i's are literals. Literals L_i are called *premises* and the atom A is called the *head* of the clause. Conforming to a standard convention, conjunctions are replaced by commas and therefore clauses are written in the form

$$A \leftarrow L_1, \ldots, L_m.$$

Logic programs constituting parts of deductive databases will be also called *database programs*. A program P is *positive* (or definite or Horn) if all of its clauses contain only positive premises. A positive program P without function symbols is called a *datalog* program and a function-less program which is not positive is called a *datalog program with negation* (*datalog*$\neg$).

If P is a program then, unless stated otherwise, we will assume that the alphabet $\mathcal{A}$ used to write P consists precisely of all the constant, predicate and function symbols that explicitely *appear* in P and thus $\mathcal{A} = \mathcal{A}_P$ is completely determined[2] by the program P. We can then talk about the first order *language* $\mathcal{L} = \mathcal{L}_P$ *of the program* P and the *Herbrand base* $\mathcal{H} = \mathcal{H}_P$ *of the program*.

It is always possible, by means of a simple transformation of a given database program P, to assume that the set S of all predicate symbols in the alphabet $\mathcal{A}$ is decomposed into two disjoint subsets S_E and S_I of the so called *extensional* and *intensional* predicates so that if P_E is the set of all clauses from P, whose heads belong to S_E and if P_I is the remaining set of clauses, namely those clauses whose heads belong to S_E, then P_E consists entirely of ground atomic formulae and P_I does not contain any ground atomic formulae. The subprogram P_E is called the *extensional part* of P and P_I is the *intensional part* of P. This division is quite important for data menagement and query processing. The intensional part P_I is usually assumed to be relatively small and fairly static and it represents the *deductive* component of the program. The extensional part P_E is normally relatively large, subject to extensive changes (insertions, deletions etc.) and represents the *relational* component of the database. From the semantic point of view, however, the distinction between extensional and intensional parts is irrelevant.

4 Model Theory

Throughout most of the paper, we consider only *Herbrand* interpretations and models; the *equality predicate* $=$ (if present) is interpreted as identity. In Section 6.8 we discuss the role and importance of non-Herbrand interpretations.

Definition 4.1 *By a* (2-valued) Herbrand interpretation I *of* $\mathcal{L}$ *we mean any set of ground atoms, i.e., any subset of the Herbrand base* $\mathcal{H}$.

[2] If there are no constants in P then one is added to the alphabet.

Clearly, any (2-valued) interpretation I can be equivalently viewed as a tuple $< T, F >$, where T and F are disjoint subsets of the Herbrand base $\mathcal{H}$, T is the set of atoms which are true in I (i.e., those that belong to I) and F is the set of atoms which are false in I (i.e., those that do not belong to I). Interpretations I defined above are called *2-valued* because they satisfy the condition $\mathcal{H} = T \cup F$ and thus assign to every ground atom either the value *true* or *false*.

Interpretations or, more precisely, models of a given theory T can be thought of as "possible worlds" representing possible states of our knowledge about T. Since, at any given moment, our knowledge about the world is likely to be incomplete, we need the ability to describe possible worlds in which some facts are neither true nor false but rather *unknown or undefined*. That is why we need *3-valued* interpretations and models to describe possible states of our knowledge.

The notion of a 3-valued interpretation is a straightforward extension of the 2-valued definition obtained by removing the requirement that $\mathcal{H} = T \cup F$.

Definition 4.2 *By a* 3-valued Herbrand interpretation *I of the language* $\mathcal{L}$ *we mean any pair* $< T; F >$*, where T and F are disjoint subsets of the Herbrand base* $\mathcal{H}$*. The set T contains all ground atoms true in I, the set F contains all ground atoms false in I and the truth value of the remaining atoms in* $U = \mathcal{H} - (T \cup F)$ *is unknown (or undefined).*

Thus a 3-valued interpretation is 2-valued iff $\mathcal{H} = T \cup F$ or, equivalently, iff $U = \emptyset$. The following proposition is obvious.

Proposition 4.1 *Any interpretation* $I =< T, F >$ *can be equivalently viewed as a function* $I : \mathcal{H} \rightarrow \{0, \frac{1}{2}, 1\}$*, from the Herbrand base* $\mathcal{H}$ *to the 3-element set* $\mathcal{V} = \{0, \frac{1}{2}, 1\}$*, defined by:*

$$I(A) = \begin{cases} 0, & \text{if } A \in F \\ \frac{1}{2}, & \text{if } A \in U \\ 1, & \text{if } A \in T. \end{cases}$$

An interpretation I is 2-valued iff I maps $\mathcal{H}$ *into the 2-element set {0,1}.*

Unless stated otherwise, 3-valued interpretations will be viewed as 3-valued functions into $\mathcal{V}$. The function (interpretation) $I : \mathcal{H} \rightarrow \mathcal{V}$ can be be recursively extended to the truth valuation $\hat{I} : \mathcal{C} \rightarrow \mathcal{V}$ defined on the set $\mathcal{C}$ of all closed formulae of the language.

Definition 4.3 *If I is an interpretation, then the* truth valuation $\hat{I}$ *corresponding to I is a function* $\hat{I} : \mathcal{C} \rightarrow \mathcal{V}$ *from the set* $\mathcal{C}$ *of all (closed) formulae of the language to* $\mathcal{V}$ *recursively defined as follows:*

- *If A is a ground atom, then* $\hat{I}(A) = I(A)$.
- *If S is a formula then* $\hat{I}(\neg S) = 1 - \hat{I}(S)$.

- *If S and V are formulae, then*

$$\begin{aligned} \hat{I}(S \wedge V) &= min(\hat{I}(S), \hat{I}(V)); \\ \hat{I}(S \vee V) &= max\{\hat{I}(S), \hat{I}(V)\}; \\ \hat{I}(V \leftarrow S) &= \begin{cases} 1, & \text{if } \hat{I}(V) \geq \hat{I}(S) \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

- *For any formula $S(x)$ with one unbounded variable x:*

$$\begin{aligned} \hat{I}(\forall x\ S(x)) &= min\{\hat{I}(S(A)) : A \in \mathcal{H}\}; \\ \hat{I}(\exists x\ S(x)) &= max\{\hat{I}(S(A)) : A \in \mathcal{H}\}; \end{aligned}$$

  where the maximum (resp. minimum) of an empty set is defined as 0 (resp. 1).

Truth valuations assign to every formula F a number $0, \frac{1}{2}$ or 1, which reflects the *degree of truth* of F, ranging from the lowest, namely *false* (0), through *unknown* ($\frac{1}{2}$), to the highest, namely *true* (1).

Remark 4.1 Our definition of the truth valuation $\hat{I}$ for the connectives $\vee, \wedge, \neg$ and for the quantifiers $\forall, \exists$ uses the so called (strong) Kleene tables [Kle52] and The truth valuation for the connective $\leftarrow$ is modeled after the approach proposed in [Fit85] and later applied in [Kun87]. There the connective $\leftrightarrow$ is introduced and used to to define 3-valued extensions of the Clark predicate completion semantics. The motivation behind our definition of the truth valuation for the implication $V \leftarrow S$ is that the truth value of the consequent V is supposed to be greater than or equal to the truth value of the premise S. In this respect, our approach essentially coincides with the approach proposed in [Fit85] and later applied in [Kun87]. However, since the primary goal of [Fit85, Kun87] was to define 3-valued extensions of the Clark predicate completion semantics, Fitting and Kunen considered 3-valued models of the *Clark completion comp*(P) of a logic program P rather than 3-valued models of the program P itself. They also considered a different ordering of truth values based on the *degree of information* rather than on the *degree of truth*. Under this ordering, the 'unknown' value is less than both values 'true' and 'false', with 'true' and 'false' being incompatible. As we will see below, this discrepancy immediately leads to different notions of minimality of a model.

Definition 4.4 *A* theory *over $\mathcal{L}$ is a (finite or infinite) set of closed formulae of $\mathcal{L}$. An interpretation I is a (2-valued or 3-valued)* model *of a theory R if $\hat{I}(S) = 1$, for all formulae S in R.*

Clearly, every program P is a theory. The following proposition is immediate.

Proposition 4.2 *An (Herbrand) interpretation M is a model of a program P if and only if for every ground instance*

$$A \leftarrow L_1, \ldots, L_m$$

of a program clause we have

$$\hat{M}(A) \geq min\{\hat{M}(L_i) : i \leq m\}.$$

Thus, M is a model of a program if and only if the degree of truth of the head of every clause is at least as high as the degree of truth of the conjunction of its premises. By a *ground instantiation* of a logic program P we mean the (possibly infinite) theory consisting of all ground instances of clauses from P.

Corollary 4.1 *An (Herbrand) interpretation M is a model of a program P if and only if it is a model of its ground instantiation.*

The above corollary shows that for model-theoretic purposes (as long as only Herbrand interpretations are considered) one can identify any program P with its ground instantiation. Whenever convenient, we will assume, without further mention, that the program P has already been instantiated.

We now define additional connectives $(\leftrightarrow, \Leftarrow, \Leftrightarrow)$ in the usual way

$$\begin{aligned} S \leftrightarrow V &\equiv (S \leftarrow V) \wedge (V \leftarrow S); \\ V \Leftarrow S &\equiv V \vee \neg S; \\ S \Leftrightarrow V &\equiv (S \Leftarrow V) \wedge (V \Leftarrow S); \end{aligned}$$

where V and S are any two formulae.

Notice, that, although the two implications "V $\leftarrow S$" and "$V \Leftarrow S$" have the same 2-valued models, in general, they have different 3-valued models. Indeed, $\hat{M}(V \leftarrow V) = 1$ no matter what the truth value of V under $\hat{M}$ is, but $\hat{M}(V \Leftarrow V) = \frac{1}{2}$ if $\hat{M}(V) = \frac{1}{2}$. This reflects the fact that in 3-valued logic we have at least two natural notions of implication, which are applicable in different contexts. Similar remarks apply to the two associated equivalence connectives $\leftrightarrow$ and $\Leftrightarrow$.

Observe, however, that in the definition of a program clause we use the implication symbol $\leftarrow$ rather than $\Leftarrow$, because we want the trivial clause $S \leftarrow S$ to be satisfied in every interpretation, regardless of the truth value of S.

Definition 4.5 *[Prz89a] If I and J are two interpretations then we say that* $I \preceq J$ *if*

$$I(A) \leq J(A) \quad \textit{(or, equivalently, } \hat{I}(A) \leq \hat{J}(A))$$

for any ground atom A. If $\mathcal{I}$ *is a collection of interpretations, then an interpretation* $I \in \mathcal{I}$ *is called* minimal *in* $\mathcal{I}$ *if there is no interpretation* $J \in \mathcal{I}$ *such that* $J \preceq I$ *and* $J \neq I$. *An interpretation I is called* least *in* $\mathcal{I}$ *if* $I \preceq J$, *for any other interpretation* $J \in \mathcal{I}$. *A model M of a theory R is called* minimal *(resp.* least*) if it is minimal (resp. least) among all models of R.*

Proposition 4.3 *If $I =< T; F >$ and $I' =< T'; F' >$ are two interpretations, then $I \preceq I'$ iff $T \subseteq T'$ and $F \supseteq F'$. In particular, for 2-valued interpretations, $I \preceq I'$ iff $I \subseteq I'$.*

Thus $I \preceq I'$ if and only if I has no more true facts and no less false facts than I'. This means that minimal and least models of a theory R minimize the *degree of truth* of their atoms, by minimizing the set T of true atoms and maximizing the set F of false atoms F. In particular, the least interpretation in the set of all interpretations is given by $I =< \emptyset, \mathcal{H} >$.

As we mentioned above, [Fit85] considers a different ordering of truth values based on the *degree of information* rather than on the *degree of truth*. Under this ordering, the 'unknown' value is less than both values 'true' and 'false', with 'true' and 'false' being incompatible. This immediately leads to a different ordering between interpretations and to different notions of minimal and least models.

Definition 4.6 *[Fit85] If $I =< T; F >$ and $I' =< T'; F' >$ are two interpretations, then we say that $I \preceq_F I'$ iff $T \subseteq T'$ and $F \subseteq F'$. We call this ordering the* F-ordering. *If $\mathcal{I}$ is a collection of interpretations, then an interpretation $I \in \mathcal{I}$ is called* F-minimal *in $\mathcal{I}$ if there is no interpretation $J \in \mathcal{I}$ such that $J \preceq_F I$ and $J \neq I$. An interpretation I is called* F-least *in $\mathcal{I}$ if $I \preceq_F J$, for any other interpretation $J \in \mathcal{I}$. A model M of a theory R is called* F-minimal *(resp.* F-least*) if it is F-minimal (resp. F-least) among all models of R.*

In particular, the F-least interpretation in the set of all interpretations is given by $I =< \emptyset, \emptyset >$. The notions of *F-minimal* and *F-least* models are different from the notions of *minimal* and *least* models (see Definition 4.5). While minimal and least models of a theory R minimize the *degree of truth* of their atoms, by minimizing the set T of true atoms and maximizing the set F of false atoms F, F-minimal and F-least models minimize the *degree of information* of their atoms, by jointly minimizing the sets T and F of atoms which are either true or false and thus maximizing the set U of unknown atoms. For example, the F-least model of the program $p \leftarrow p$ is obtained when p is undefined, while the least model of P is obtained when p is false. As it will be seen in the sequel, this distinction reflects fundamental differences between the semantics based on Clark's completion and model-theoretic semantics, such as the least model semantics, perfect model semantics or well-founded semantics, which are the main topic of this paper.

5 Fixed Points

Declarative semantics of logic programs is often defined using fixed points of some natural operators Ψ acting on *ordered* sets of interpretations. Suppose $\leq$ is an ordering on the set $\mathcal{I}$ of interpretations of a given language, $\mathcal{J}$ is a subset of $\mathcal{I}$ and Ψ is an *operator* $\Psi : \mathcal{I} \rightarrow \mathcal{I}$ on $\mathcal{I}$.

Definition 5.1 *The operator* Ψ *is called* monotone *if* $I \leq J$ *implies* $\Psi(I) \leq \Psi(J)$*, for any* $I, J \in \mathcal{I}$*. An interpretation* $I \in \mathcal{I}$ *is a* fixed point *of* Ψ *if* $\Psi(I) = I$*. By* the least upper bound $\Sigma\mathcal{J}$ of $\mathcal{J}$ *(resp.* the greatest lower bound $\Pi\mathcal{J}$ of $\mathcal{J}$*) we mean an interpretation* $I \in \mathcal{I}$ *such that* $J \leq I$*, for any* $J \in \mathcal{J}$ *and* $J \leq J'$ *for any other* J' *with this property (resp.* $I \leq J$*, for any* $J \in \mathcal{J}$ *and* $J' \leq J$ *for any other* J' *with this property). By the* smallest *interpretation (under the given ordering) we mean an interpretation* I_0 *such that* $I_0 \leq I$*, for any other interpretation* I*.*

Least fixed points of monotone operators Ψ are often generated by *iterating* the operator Ψ starting from the *smallest interpretation* I_0 and obtaining the (possibly transfinite) sequence:

$$\begin{aligned} \Psi^{\uparrow 0} &= I_0; \\ \Psi^{\uparrow \alpha+1} &= \Psi(\Psi^{\uparrow \alpha}); \\ \Psi^{\uparrow \lambda} &= \Sigma_{\alpha<\lambda}\Psi^{\uparrow \alpha}; \end{aligned}$$

for limit ordinals λ. Clearly, an iteration $\Psi^{\uparrow \alpha}$ is a fixed point of Ψ if and only if

$$\Psi^{\uparrow \alpha} = \Psi^{\uparrow \alpha+1}.$$

In the sequel we will consider two principal orderings among interpretations, namely the *standard* ordering $\preceq$ (see Definition 4.5) and the *F-ordering* $\preceq_F$ (see Definition 4.6). Operators acting on sets of interpretations ordered by the standard ordering, will be denoted by Ψ or Θ, while those acting on sets of interpretations ordered by the F-ordering, will be denoted by Φ or Ω. Recall that $I_0 =< \emptyset, \mathcal{H} >$ (resp. $I_0 =< \emptyset, \emptyset >$) is the smallest (resp. F-smallest) interpretation in the set of all interpretations ordered by $\preceq$ (resp. $\preceq_F$).

For a subset $\mathcal{J}$ of $\mathcal{I}$, we will denote by $\Sigma\mathcal{J}$ (resp. $\Pi\mathcal{J}$) the least upper bound (resp. the greatest lower bound) of $\mathcal{J}$ with respect to $\preceq$. Similarly, we will denote by $\Sigma_F\mathcal{J}$ (resp. $\Pi_F\mathcal{J}$) the least upper bound (resp. the greatest lower bound) of $\mathcal{J}$ with respect to $\preceq_F$.

Observe, that if $\mathcal{J} = \{J_s : s \in S\}$, with $J_s =< T_s, F_s >$, then:

$$\begin{aligned} \Sigma\mathcal{J} &= < \bigcup_{s\in S} T_s, \bigcap_{s\in S} F_s >; \\ \Pi\mathcal{J} &= < \bigcap_{s\in S} T_s, \bigcup_{s\in S} F_s >; \\ \Sigma_F\mathcal{J} &= < \bigcup_{s\in S} T_s, \bigcup_{s\in S} F_s >; \\ \Pi_F\mathcal{J} &= < \bigcap_{s\in S} T_s, \bigcap_{s\in S} F_s > . \end{aligned}$$

Although $\Sigma\mathcal{J}$, $\Pi\mathcal{J}$ and $\Pi_F\mathcal{J}$ are always well-defined interpretations, $\Sigma_F\mathcal{J} =< T, F >$ may not be an interpretation, because the sets T and F may not be

disjoint. However, $\Sigma_F \mathcal{J}$ is always an interpretation, provided that $\mathcal{J}$ is an *F-directed* set of interpretations, i.e., such that for any $J, J' \in \mathcal{J}$ there is a $J'' \in \mathcal{J}$ satisfying $J \preceq_F J''$ and $J' \preceq_F J''$.

6 Declarative Semantics of Deductive Databases and Logic Programs

A precise meaning or *semantics* must be associated with any logic or database program P in order to provide a declarative specification of the program. *Declarative semantics* provides a mathematically precise definition of the meaning of the program in a manner, which is independent of procedural considerations, context-free, and easy to manipulate, exchange and reason about.

Procedural semantics of a logic program, on the other hand, usually is given by providing a procedural mechanism that, at least in theory and perhaps under some additional assumptions, is capable of providing answers to a wide class of queries. The performance of such a mechanism (in particular, its correctness) is evaluated by comparing its behavior to the *specification* provided by the declarative semantics. Without a proper declarative semantics the user needs an intimate knowledge of procedural aspects in order to write correct programs.

Finding a suitable declarative or intended semantics is one of the most important and difficult problems in logic programming and deductive databases. The importance of this problem stems from the declarative character of logic programs and deductive databases, whereas its difficulty can be largely attributed to the fact that there does not exist a precisely defined set of conditions that a 'suitable' semantics should satisfy. While all researchers seem to agree that any semantics SEM(P) must reflect the intended meaning of a program or a database and also be suitable for mechanical computation, there is no agreement as to which semantics best satisfy these criteria.

One thing, however, appears to be clear. Logic programs and deductive databases must be as easy to write and comprehend as possible, free from excessive amounts of explicit negative information and as close to natural discourse as possible. In other words, the declarative semantics of a program or a database must be determined more by its *commonsense meaning* than by its purely logical contents. For example, given the information that 1 is a natural number and that $n + 1$ is a natural number if so is n, we should be able to derive a non-monotonic or commonsense conclusion that neither 0 nor *Mickey Mouse* is a natural number. Similarly, from a database of information about teaching assignments, which only shows that John teaches Pascal and Prolog this semester, it should be possible to reach a common sense conclusion that John does not teach Calculus. Clearly, none of these facts follow logically from our assumptions.

The intended semantics must therefore be *non-monotonic*, i.e., SEM(P)

cannot monotonically increase with the theory P. For example, after learning (adding to P) the fact that 0 is also a natural number, we have to withdraw the previously reached conclusion to the contrary. Consequently, the problem of finding a suitable semantics for logic programs and deductive databases can be viewed as the problem of finding a suitable non-monotonic formalization of the type of reasoning used in logic programs and deductive databases. We briefly discuss the relationship between the proposed semantics and non-monotonic formalisms in Section 7. The problem of a suitable semantics for *negation* is just a special case of the more general problem of the intended semantics of a program or a database.

Declarative semantics SEM(P) of a program can be specified in various ways, among which the following two are most common. One that can be called *proof-theoretic*, associates with P its first order completion $COMP(P)$ (e.g., $COMP(P)$ can be P itself or the Clark predicate completion comp(P) of P). A formula V is said to be *implied* by the semantics SEM(P) if and only if it is logically implied by the completion

$$COMP(P) \models V,$$

i.e., if V is satisfied in all 2-valued (Herbrand or not) models of $COMP(P)$.

Another method of defining the declarative semantics SEM(P) of a program is *model-theoretic*. The semantics is determined by choosing a set $MOD(P)$ of *intended* models of P (in particular, one intended model M_P). For example, MOD(P) can be the set of all minimal models of P or the unique least model of P. A formula V is said to be *implied* by the semantics SEM(P) if and only if it is satisfied in all intended models:

$$MOD(P) \models V \quad (\text{in particular}, \quad M_P \models V).$$

Observe, that the proof-theoretic approach can be viewed as a *special case* of the model-theoretic approach. Other approaches to defining the declarative semantics are possible, e.g., a combination of proof-theoretic and model-theoretic methods has been used in [Kun87, Fit85].

It is important to point out that the terms *model-theoretic* and *proof-theoretic* are used here in a different sense than. e.g., in [Rei84, GMN84]. In those papers a model-theoretic approach to deductive databases was described to mean *viewing a deductive database as an interpretation* and it was contrasted with a proof-theoretic approach, which means *viewing a deductive database as a first order theory*. We always view a deductive database or a logic program P as a first order theory, but using a proof-theoretic approach we define its *semantics* by means of a first order completion of P, whereas using a model-theoretic approach we define the semantics by selecting one or more intended models of P.

In the remainder of this section we will review and discuss some of the proposed declarative semantics for logic programs and deductive databases. Our

discussion is not meant to be exhaustive and the selection of semantics clearly reflects the authors' preferences. We will begin with a discussion of the Clark predicate completion semantics and its 3-valued extensions. Subsequently, we will concentrate on model-theoretic semantics, including the least model semantics, the perfect model semantics, the stable model semantics, the weakly perfect model semantics and the well-founded semantics. We finish with a brief discussion of the role and importance of semantics based on *non-Herbrand models*. Throughout our presentation, we will strive to explore relationships existing between various approaches.

6.1 Clark's Predicate Completion Semantics

The most commonly used declarative semantics of logic programs, although less popular in the context of deductive databases, is based on the so called *Clark predicate completion comp(P)* of a logic program P [Cla78, Llo84].

Clark's completion of P is obtained by first rewriting every clause in P of the form:

$$q(K_1, \ldots, K_n) \leftarrow L_1, \ldots, L_m,$$

where q is a predicate symbol and $K_1, \ldots, K_n$ are terms containing variables $X_1, \ldots, X_k$, as a clause

$$q(T_1, \ldots, T_n) \leftarrow V,$$

where T_i's are variables,

$$V = \exists X_1, \ldots, X_k \ (T_1 = K_1 \wedge \ldots \wedge T_n = K_n \wedge L_1 \wedge \ldots \wedge L_m)$$

and then replacing, for every predicate symbol q in the alphabet, the (possibly empty[3]) set of all clauses

$$\begin{array}{lcl} q(T_1, \ldots, T_n) & \leftarrow & V_1 \\ & \ldots & \\ & \ldots & \\ q(T_1, \ldots, T_n) & \leftarrow & V_s \end{array}$$

with q appearing in the head, by a single universally quantified logical equivalence

$$q(T_1, \ldots, T_n) \leftrightarrow V_1 \vee \ldots \vee V_s.$$

Finally, the obtained theory is augmented by the so called *Clark's Equality Axioms* (see Section 6.8), which include unique names axioms and axioms for equality. These axioms are essential when considering non-Herbrand models of Clark's completion.

[3] If there are no clauses involving the head $q(T_1, \ldots, T_n)$, then the corresponding disjunction is empty and thus always false. The resulting completion contains therefore a universal negation of $q(T_1, \ldots, T_n)$.

Clark's approach is mathematically elegant and founded on a natural idea that in common discourse we often tend to use 'if' statements, when we really mean 'iff' statements. For example, we may use the following program P_1 to describe natural numbers:

$$\begin{array}{rcl} natural\_number(0) & & \\ natural\_number(succ(X)) & \leftarrow & natural\_number(X). \end{array}$$

The above theory P_1 is rather weak. It does not even imply that, say, *Mickey Mouse* is *not* a natural number. This is because, what we really have in mind is

$$natural\_number(T) \quad \longleftrightarrow \quad \exists X \ (T = 0 \vee (T = succ(X) \wedge natural\_number(X)))$$

which is in fact Clark's completion of P_1 and it indeed implies

$$\neg natural\_number(MickeyMouse).$$

Unfortunately, Clark's predicate completion semantics has some serious drawbacks. One of them is the fact that *Clark's completion is often inconsistent*, i.e., it may not have any *2-valued* (Herbrand or not) models, in which case Clark's semantics is undefined. For example, Clark's completion of the program $p \leftarrow \neg p$ is $p \leftrightarrow \neg p$, which is inconsistent. The situation can be even worse, e.g., Clark's completion of the program P_2:

$$p \leftarrow \neg q, \neg p$$

is:

$$\begin{array}{rcl} p & \leftrightarrow & \neg q, \neg p \\ \neg q & & \end{array}$$

which is inconsistent. However, after adding to P_2 a 'meaningless' clause $q \leftarrow q$ its completion becomes:

$$\begin{array}{rcl} p & \leftrightarrow & \neg q, \neg p \\ q & \leftrightarrow & q \end{array}$$

which has a unique 2-valued model in which q is true and p is false. On the other hand, after adding to P another 'meaningless' clause $p \leftarrow p$ its completion becomes:

$$\begin{array}{rcl} p & \leftrightarrow & p \vee (\neg q \wedge \neg p) \\ \neg q & & \end{array}$$

which has a unique, yet different, 2-valued model in which q is false and p is true.

6.1.1 Three-Valued Extensions

[Fit85] showed that the inconsistency problem for Clark's semantics, as well as some other related problems, can be elegantly eliminated by considering *3-valued* Herbrand models of the Clark predicate completion $comp(P)$, rather than 2-valued models only.

Theorem 6.1 *[Fit85] Clark's completion $comp(P)$ of* any *logic program P always has at least one 3-valued Herbrand model. Moreover, among all 3-valued models of $comp(P)$ there is exactly one* F-least *model M_P.*

This result gave rise to Fitting's 3-valued extension of Clark's semantics.

Definition 6.1 (Fitting's Semantics) *[Fit85]* Fitting's 3-valued extension of the Clark predicate completion semantics *is the semantics determined by the unique intended model M_P or, equivalently, by the set $MOD(P)$ of intended models, consisting of all* 3-valued Herbrand models of $comp(P)$.

For example, the program P_2 defined before has a unique 3-valued model in which q is false and p is undefined. Fitting also provided an elegant fixed-point characterization of 3-valued models of $comp(P)$.

Definition 6.2 (The Fitting Operator) *[Fit85] Suppose that P is a logic program. The* Fitting operator $\Phi : \mathcal{I} \rightarrow \mathcal{I}$ *on the set $\mathcal{I}$ of all 3-valued interpretations of $comp(P)$ is defined as follows. If $I \in \mathcal{I}$ is an interpretation of $comp(P)$ and A is a ground atom then $\Phi(I)$ is an interpretation given by*[4]*:*

(i) $\Phi(I)(A) = 1$ *if there is a clause $A \leftarrow L_1, \ldots, L_n$ in P such that $\hat{I}(L_i) = 1$, for all $i \leq n$;*

(ii) $\Phi(I)(A) = 0$ *if for every clause $A \leftarrow L_1, \ldots, L_n$ in P there is an $i \leq n$ such that $\hat{I}(L_i) = 0$;*

(iii) $\Phi(I)(A) = \frac{1}{2}$ *, otherwise.*

Theorem 6.2 *[Fit85] An interpretation I of $comp(P)$ is a model of $comp(P)$ if and only if it is a fixed point of the operator Φ. In particular, M_P is the F-least fixed point of Φ.*

Moreover, the model M_P can be obtained by iterating the operator Φ, namely, the sequence $\Phi^{\uparrow\alpha}$ of iterations[5] *of Φ is monotonically increasing and it has a fixed point*

$$\Phi^{\uparrow\lambda} = M_P.$$

[4] According to the conventions adopted in Section 4, P is assumed to be instantiated and interpretations are viewed as 3-valued functions.

[5] According to the convention from Section 5, Φ is defined on the F-ordered set of interpretations and the iteration begins from the F-smallest interpretation $I = < \emptyset, \emptyset >$.

Kunen [Kun87] showed that the set of formulae implied by Fitting's semantics is not recursively enumerable and he proposed the following modification of Fitting's approach.

Definition 6.3 (Kunen's Semantics) *[Kun87]* Kunen's 3-valued extension of the Clark predicate completion semantics *is the semantics determined by the set* $MOD(P)$ *of intended models, consisting of* all 3-valued (Herbrand or not) models of $comp(P)$.

Kunen showed that his semantics is recursively enumerable and closely related to the Fitting operator Φ.

Theorem 6.3 *[Kun87] A closed formula* V *is implied by Kunen's 3-valued extension of the Clark predicate completion semantics if and only if it is satisfied in at least one finite iteration* $\Phi^{\uparrow n}$ *of the Fitting operator* Φ, $n = 0, 1, 2, \ldots$. *Moreover, the set of formulae implied by this semantics is recursively enumerable.*

It is easy to see that Fitting's semantics is stronger than Kunen's semantics, i.e., any closed formula implied by Kunen's semantics is also implied by Fitting's semantics.

6.1.2 Drawbacks of Clark's Completion Semantics

Unfortunately, Clark's predicate completion does not always result in a satisfactory semantics. For many programs, it leads to a semantics which appears *too weak*. This problem applies both to standard Clark's semantics as well as to its 3-valued extensions and it has been extensively discussed in the literature (see e.g. [She88, She84, Prz89c, VGRS90]). We illustrate it on the following three examples.

Example 6.1 Suppose that to the program P_1 defined before we add a seemingly meaningless clause:

$$natural\_number(X) \leftarrow natural\_number(X).$$

It appears that the newly obtained program P_1' should have the same semantics. However, Clark's completion of the new program P_1' is:

$$natural\_number(T) \longleftrightarrow$$

$$(natural\_number(T) \vee T = 0 \vee \exists X\ (T = succ(X) \wedge natural\_number(X)))$$

from which it no longer follows that $MickeyMouse$ (or anything else, for that matter) is *not* a natural number.

Example 6.2 [Van Gelder] Suppose, that we want to describe which vertices in a graph are reachable from a given vertex a. We could write the following program P_3:

$$\begin{aligned} edge(a,b) & \\ edge(c,d) & \\ edge(d,c) & \\ reachable(a) & \\ reachable(X) & \leftarrow reachable(Y), edge(Y,X). \end{aligned}$$

We clearly expect vertices c and d not to be reachable. However, Clark's completion of the predicate 'reachable' gives only

$$reachable(X) \longleftrightarrow (X = a \vee \exists Y\ (reachable(Y) \wedge edge(Y,X)))$$

from which such a conclusion again cannot be derived. Here, the difficulty is caused by the existence of symmetric clauses $edge(c,d)$ and $edge(d,c)$.

Example 6.3 Suppose that program P_4 is given by the following clauses:

$$\begin{aligned} bird(tweety) & \\ fly(X) & \leftarrow bird(X), \neg abnormal(X) \\ abnormal(X) & \leftarrow irregular(X) \\ irregular(X) & \leftarrow abnormal(X). \end{aligned}$$

The last two clauses merely state that irregularity is synonymous with abnormality. Based on the fact that nothing leads us to believe that tweety is abnormal, we are justified to expect that tweety flies, but Clark's completion of P_4 yields

$$\begin{aligned} fly(T) & \longleftrightarrow (bird(T) \wedge \neg abnormal(T)) \\ abnormal(T) & \longleftrightarrow irregular(T), \end{aligned}$$

from which it does not follow that *anything* flies. On the other hand, without the last two clauses (or without just one of them) Clark's semantics produces correct results.

The above described behavior of Clark's completion is bound to be confusing for a thoughtful logic programmer, who may very well wonder why, for example, the addition of a seemingly harmless statement "$natural\_number(X) \leftarrow natural\_number(X)$" should change the meaning of the first program. The explanation that will most likely occur to him will be *procedural* in nature, namely, the fact that the above added clause may lead to a loop. But it was the idea of replacing *procedural* programming by *declarative* programming, that

brought about the concept of logic programming and deductive databases in the first place, and therefore it seems that such a procedural explanation should be flatly rejected.

Some of the problems mentioned above are caused by the difficulties with the representation of transitive closures when using Clark's semantics (e.g., in the program P_3). Recently, [Kun88] formally showed that Clark's semantics is not sufficiently expressive to naturally represent transitive closures.

In the following sections we discuss model-theoretic approaches to declarative semantics of logic programs which attempt to avoid the drawbacks of Clark's semantics discussed above.

6.2 Least Model Semantics

The model-theoretic approach is particularly well-understood in the case of *positive logic programs*. In this section we assume that all interpretations are 2-valued.

Example 6.4 Suppose that our program P consists of clauses:

$$\begin{aligned} able\_mathematician(X) &\leftarrow physicist(X) \\ physicist(einstein) & \\ businessman(iacocca). & \end{aligned}$$

This program has several different models, the largest of which is the model in which both Einstein and Iacocca are at the same time businessmen, physicists and good mathematicians. This model hardly seems to correctly describe the intended meaning of P. Indeed, there is nothing in this program to imply that Iacocca is a physicist or that Einstein is a businessman. In fact, we are inclined to believe that the lack of such information indicates that we can assume the contrary.

The program also has the unique *least* model M_P:

$$\{physicist(einstein), businessman(iacocca), able\_mathematician(einstein)\},$$

in which only Einstein is a physicist and good mathematician and only Iacocca is a businessman. This model seems to correctly reflect the semantics of P, at the same time incorporating the classical case of the closed-world assumption [Rei78]: if no reason exists for some positive statement to be true, then we are allowed to infer that it is false.

It turns out that the existence of the unique least model M_P is the property shared by *all* positive programs.

Theorem 6.4 *[VEK76] Every positive logic program P has a unique least (Herbrand) model M_P.*

This important result led to the definition of the so called least model semantics for positive programs.

Definition 6.4 (Least Model Semantics) *[VEK76] By the* least model semantics *of a positive program P we mean the semantics determined by the least Herbrand model M_P of P.*

The least Herbrand model semantics is very intuitive and it seems to properly reflect the intended meaning of positive logic programs. The motivation behind this approach is based on the idea that we should minimize positive information as much as possible, limiting it to facts explicitely implied by P, and making everything else false. In other words, the least model semantics is based on a natural form of the *closed world assumption.*

The least model semantics avoids the drawbacks of the Clark predicate completion discussed in the previous section. For example, the least Herbrand model M_P of the programs P_1 and P_1' given above is:

$$\{natural\_number(0), natural\_number(succ(0)), natural\_number(succ(succ(0))), \ldots\}$$

which is exactly what we intended. Similarly, the least Herbrand model M_P of the program P_3 above is:

$$\{edge(a,b), edge(c,d), edge(d,c), reachable(a), reachable(b)\},$$

which is again exactly what we would expect.

Least model semantics also has a natural fixed point characterization. First we define the Van Emden-Kowalski immediate consequence operator $\Psi : \mathcal{I} \rightarrow \mathcal{I}$ on the set $\mathcal{I}$ of all interpretations of P (ordered by $\preceq$).

Definition 6.5 (The Van Emden-Kowalski Operator) *[VEK76] Suppose that P is a positive logic program, $I \in \mathcal{I}$ is an interpretation of P and A is a ground atom. Then $\Psi(I)$ is an interpretation given by:*

(i) $\Psi(I)(A) = 1$ *if there is a clause* $A \leftarrow A_1, \ldots, A_n$ *in P such that* $I(A_i) = 1$, *for all* $i \leq n$;

(ii) $\Psi(I)(A) = 0$, *otherwise.*

Theorem 6.5 *[VEK76] The Van Emden-Kowalski operator Ψ has the least fixed point, which coincides with the least model M_P.*

Moreover, the model M_P can be obtained by iterating ω times the operator Ψ, namely, the sequence $\Psi^{\uparrow n}$, $n = 0, 1, 2, \ldots, \omega$, of iterations[6] of Ψ is monotonically increasing and it has a fixed point

$$\Psi^{\uparrow \omega} = M_P.$$

[6] With respect to the standard ordering $\preceq$ of interpretations and beginning from the smallest interpretation $< \emptyset, \mathcal{H} >$. Here ω denotes the first infinite ordinal.

The least model semantics is strictly stronger than Clark's semantics:

Theorem 6.6 *Suppose that P is a positive logic program. If a closed formula is implied by the Clark predicate completion semantics (or by one of its 3-valued extensions) then it is also implied by the least model semantics.*

The only serious, drawback of the least model semantics seems to be the fact that it is well defined for a very restrictive class of programs. Programs which are not positive, in general, do not have least models. For example, the program $p \leftarrow \neg q$ has two minimal models $\{p\}$ and $\{q\}$, but it does not have the least model. Similarly, the program P_4 from Example 6.3 does not have the least model.

6.3 Perfect Model Semantics

As we have seen above, although the least model semantics seems suitable for the class of positive programs, it is not adequate for more general programs, allowing *negative premises* in program clauses. The inclusion of negation in program clauses increases the expressive power of logic programs and thus is of great practical importance. At the same time, the problem of finding a suitable semantics for programs with negation becomes much more complex. In this section we discuss the perfect model semantics, which extends the least model semantics to a wider class of logic programs. Throughout most of this section by an interpretation (model) we mean a *2-valued* interpretation (model).

Example 6.5 Suppose that we know that physicists are able mathematicians, whereas typical businessmen tend to avoid (advanced) mathematics in their work, unless they somehow happen to have a strong mathematical background. Suppose also that we know that Iacocca is a businessman and that Einstein is a physicist. We can express these facts using a logic program as follows:

$$avoids\_math(X) \leftarrow businessman(X), \neg able\_mathematician(X) \quad (1)$$

$$able\_mathematician(X) \leftarrow physicist(X) \quad (2)$$

$$businessman(iacocca) \quad (3)$$

$$physicist(einstein). \quad (4)$$

This program does not have a unique least model, but instead it has two *minimal* models. In both of them Iacocca is the only businessman, Einstein is the only physicist and he is also an able mathematician, who uses advanced mathematics. However, in one of them, say in M_1, Iacocca avoids advanced mathematics, because he is not an able mathematician and in the other, M_2, the situation is opposite and Iacocca is an able mathematician, who uses advanced mathematics in his work.

Since any intended semantics for logic programs must include some form of the closed world assumption, and thus it must in some way *minimize positive information*, it is natural to consider *minimal models* of P [Min82, BS85, McC80]

as providing the desired meaning of P. It seems clear, however, that *not both* minimal models capture the intended meaning of P. By placing negated predicate *able_mathematician(X)* among the premises of the rule, we intended to say that businessmen, in general, avoid advanced mathematics *unless* they are known to be good mathematicians. Since we have no information indicating that Iacocca is a good mathematician we are inclined to infer that he does not use advanced mathematics. Therefore, only the first minimal model M_1 seems to correspond to the intended meaning of P.

The reason for this asymmetry is easy to explain. The first clause (1) of P is logically (classically) equivalent to the clause

$$able\_mathematician(X) \vee avoids\_math(X) \leftarrow businessman(X) \qquad (5)$$

and models M_1 and M_2 are therefore also minimal models of the theory P' obtained from P by replacing (1) by (5). However, the intended meaning of these two clauses seems to be different. The clause (5) does not assign distinct *priorities* to predicates (properties) *able_mathematician* and *avoids_math* and thus treats them as equally plausible. As a result the semantics determined by the two minimal models M_1 and M_2 seems to be perfectly adequate to represent the intended meaning of P'. On the other hand, the program clause (1) intuitively seems to assign distinct *priorities for minimization* to predicates *able_mathematician* and *avoids_math*, essentially saying that the predicate *able-mathematician* has to be first assumed $false$ unless there is a compelling reason to do otherwise. We can say, therefore, that the clause (1) assigns a *higher priority* for minimization (or *falsification*) to the predicate *able_mathematician* than to the predicate *avoids_math*.

We can easily imagine the above priorities reversed. This is for instance the case in the following clause:

$$able\_mathematician(X) \quad \leftarrow \quad physicist(X), \neg avoids\_math(X)$$

which says that if X is a physicist and if we have no specific evidence showing that he avoids mathematics then we are allowed to assume that he is an able mathematician. Here, the predicate *avoids_math* has a higher priority for minimization than the predicate *able_mathematician*, i.e., it is supposed to be first assumed $false$ unless there is a specific reason to do otherwise.

Also observe, that if $B \leftarrow A$ is a clause, then minimizing B (i.e., making B false) immediately results in A being minimized, too. Consequently, A is always minimized before or at the same time when B is minimized. The above discussion leads us to the conclusion that the *syntax* of program clauses determines relative *priorities* for minimization among ground atoms according to the following rules:

I. Negative premises have *higher* priority than the heads;

II. Positive premises have priority *no less* than that of the heads.

To formalize conditions I and II, we assume that the program is already instantiated and we introduce the *dependency graph* G_P of P (cf. [ABW88, VG89b]), whose vertices are *ground atoms*, i.e., elements of the Herbrand base $\mathcal{H}$. If A and B are atoms, then there is a directed edge in G_P from B to A if and only if there is a clause in P, whose head is A and one of whose premises is either B or $\neg B$. In the latter case the edge is called *negative*.

Definition 6.6 (Priority Relation) *[ABW88] For any two ground atoms A and B in $\mathcal{H}$ we define B to have a* higher priority[7] *than A ($A < B$) if there is a directed path in G leading from B to A and passing through at least one negative edge. We call the above defined relation $<$* the priority relation *between (ground) atoms. We will write $A \leq B$ if there is a directed path from B to A.*

Analogously, we can define the *predicate priority relation* $<_P$ between predicate symbols, replacing in the above definition ground atoms by predicate symbols. Having defined the priority relation, we are prepared to define the notion of a *perfect model*. It is our goal to define a minimal model in which atoms of higher priority are *minimized* (or *falsified*) first, even at the cost of including in the model (i.e., making true in it) some atoms of lower priority. It follows, that if M is a model of P and if a new model N is obtained from M, by adding and subtracting from M some atoms, then we will consider the new model N *preferable* to M if and only if the addition of any atom A is always *justified* by the removal of a higher priority atom B (i.e. such that $A < B$). A model M of P will be considered *perfect*, if there are *no* models of P preferable to it. More formally:

Definition 6.7 (Perfect Models) *[Prz88a, Prz89c] Suppose that M and N are two distinct models of a logic program P. We say that N is* preferable *to M (briefly,* N << M*), if for every atom $A \in N - M$ there is a higher priority atom B, $B > A$, such that $B \in M - N$. We say that a model M of P is* perfect *if there are no models preferable to M. We call the relation $<<$ the* preference relation *between models.*

It is easy to prove

Theorem 6.7 *[Prz88a] Every perfect model is minimal. For positive programs the concepts of a least model and a perfect model coincide.*

Example 6.6 Only model M_1 in Example 6.5 is perfect. Indeed (using obvious abbreviations):

$$M_1 = \{physicist(e), able\_mathematician(e), businessman(i), avoids\_math(i)\}$$

$$M_2 = \{physicist(e), able\_mathematician(e), businessman(i), able\_mathematician(i)\}$$

[7] There is no consensus in the literature as to whether to describe this property as having 'higher' or 'lower' priority and, accordingly, as to whether to denote it by $A < B$ or $A > B$.

and we know that able_mathematician > avoids_math and therefore $M_1 << M_2$, while not $M_2 << M_1$. Consequently, M_1 is perfect, but M_2 is not.

Unfortunately, not every logic program has a perfect model:

Example 6.7 The program:

$$p \leftarrow \neg q \quad , \quad q \leftarrow \neg p$$

has only two minimal Herbrand models $M_1 = \{p\}$ and $M_2 = \{q\}$ and since p<q and q<p we have $M_1 << M_2$ and $M_2 << M_1$, thus none of the models is perfect.

The cause of this peculiarity is quite clear. The concept of a perfect model is based on relative priorities between ground atoms and therefore we have to be consistent when assigning those priorities to avoid priority conflicts (cycles), which could render our semantics meaningless. This observation underlies the approaches of Apt, Blair and Walker [ABW88] and Van Gelder [VG89b], who argued that *when using negation we should be referring to an already defined relation*, so that the definition is not circular, or, as Van Gelder puts it, we should avoid *negative recursion*. This idea led them to the introduction of the class of *stratified* logic programs (see also [CH85, Naq86]). The class of stratified logic programs has been later extended [Prz88a] to the class of *locally stratified* programs.

Definition 6.8 *[ABW88, VG89b, Prz88a] A logic program P is* stratifed *(resp.* locally stratified*) if it is possible to decompose the set S of all predicate symbols (resp. the Herbrand base $\mathcal{H}$) into disjoint sets $S_1, S_2, \ldots, S_\alpha, \ldots$, $\alpha < \lambda$, called* strata*, so that for every clause (resp. instantiated clause):*

$$C \leftarrow A_1, ..., A_m, \neg B_1, ..., \neg B_n$$

in P, where A's, B's and C are atoms, we have that:

(i) for every i, stratum(A_i) $\leq$ stratum(C),

(ii) for every j, stratum(B_j) < stratum(C),

where stratum(A) = α, if the predicate symbol of A belongs to S_α (resp. if the atom A belongs to S_α). Any particular decomposition $\{S_1, \ldots, S_\alpha, \ldots\}$ satisfying the above conditions is called a stratification of P *(resp.* local stratification of P*).*

In the above definition, stratification determines priority levels (strata), with lower level (stratum) denoting higher priority for minimization. For example, the program from Example 6.5 is stratified and one of its stratifications is S_1={able_mathematician}, S_2={businessman, physicist, avoids_math}.

The difference between the definitions of stratification and local stratification is that in the first case we decompose the set S of all predicate symbols, while in the second case we decompose the Herbrand base $\mathcal{H}$. Since every program can effectively refer only to a finite set of predicate symbols, stratifications can be always assumed to be finite. On the other hand, if the program uses function symbols then its Herbrand universe is infinite and its local stratifications can, in general, be infinite. The following fact is obvious:

Proposition 6.1 *Every stratified program is locally stratified.*

The next proposition characterizes (local) stratifiability.

Proposition 6.2 *[ABW88, Prz88a] A logic program P is stratified if and only if its predicate priority relation $<_P$ is a partial order*[8]*.*

A logic program P is locally stratified if and only if its priority relation $<$ is a partial order and if every increasing sequence of ground atoms under $<$ is finite[9]*.*

All programs described so far in this paper, with the exception of Example 6.7, are stratified. The program in Example 6.7 is not even locally stratified. We now present an example of a locally stratified program which is not stratified.

Example 6.8 The following program defines even numbers:

$$even(0)$$

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

Here s(X) is meant to represent the successor function on the set of natural numbers. This program is not stratified because the predicate *even* is involved in negative recursion with itself, i.e., $even <_P even$. However, P is locally stratified, because the priority ordering $<$ between ground atoms is easily seen to be a partial order and every increasing sequence of ground atoms is of the form:

$$even(s(s(s(...)))) < even(s(s(...))) < even(s(...)) < ... < even(s(0)) < even(0)$$

and therefore it must be finite.

The following basic result shows that every locally stratified program has the least model M_P with respect to the preference relation $<<$.

Theorem 6.8 *[Prz88a] Every locally stratified program P has a unique perfect model M_P. Moreover, M_P is preferred to any other model M of P, i.e., $M_P << M$, for any other model M.*

[8] By a *partial order* we mean an irreflexive and transitive relation.

[9] The last condition is only essential when the Herbrand base is infinite.

For stratified programs, models M_P have been first introduced under the name of 'natural' models in [ABW88, VG89b] and defined in terms of iterated fixed points and iterated least models. In general, a (locally) stratified program may have many stratifications, however, the notion of a perfect model is defined entirely in terms of the priority relation $<$ and thus it does not depend on a particular stratification. Now we can define the perfect model semantics of locally stratified logic programs.

Definition 6.9 (Perfect Model Semantics) *[ABW88, VG89b, Prz88a] Let P be a locally stratified*[10] *logic program. By the* perfect model semantics *of P we mean the semantics determined by the unique perfect model M_P of P.*

It follows immediately from Theorem 6.7 that for positive logic programs the perfect model semantics is in fact equivalent to the *least model semantics* and thus the perfect model semantics *extends* the least model semantics.

The following result, slightly generalizing [ABW88], shows that the perfect model semantics is strictly *stronger* than the semantics defined by Clark's completion.

Theorem 6.9 *(cf. [ABW88]) If P is a locally stratified logic program, then Clark's completion comp(P) is consistent and if a closed formula is implied by the Clark predicate completion semantics (or by one of its 3-valued extensions) then it is also implied by the perfect model semantics.*

The perfect model semantics eliminates various unituitive features of Clark's semantics discussed before. For example, the unique perfect model of the program P_4 discussed in Example 6.3 consists of:

$$\{bird(tweety),\ fly(tweety)\},$$

leading to the expected intended semantics. The perfect model semantics is actually used in two large experimental deductive database systems, namely in the LDL system, implemented at MCC [Zan88], and in the NAIL! system, which is currently under development at Stanford [MUG86].

6.3.1 Perfect Models As Iterated Fixed Points and Iterated Least Models

Least models of positive programs have been characterized as fixed points of the Van Emden-Kowalski operator. It turns out that perfect models of locally stratified programs can be also characterized as *iterated least fixed points* and as *iterated least models* of the program. In the remainder of this section we consider both 2-valued and 3-valued interpretations.

[10] Perfect model semantics can be defined for a significantly larger class of programs, but for the sake of compatibility with its extensions discussed in the following sections, we limit it to the class of locally stratified programs.

First we need a generalization of the Van Emden-Kowalski operator Ψ defined in Section 6.2. For any interpretation J we define a corresponding operator Ψ_J as follows:

Definition 6.10 (The Generalized Van Emden-Kowalski Operator) *Suppose that P is any logic program, $I, J \in \mathcal{I}$ are interpretations and A is a ground atom. Then $\Psi_J(I)$ is a (2-valued) interpretation given by:*

(i) $\Psi_J(I)(A) = 1$ *if there is a clause* $A \leftarrow L_1, \ldots, L_n$ *in* P *such that, for all* $i \leq n$, *either* $\hat{J}(L_i) = 1$ *or* L_i *is an atom and* $I(L_i) = 1$;

(ii) $\Psi_J(I)(A) = 0$, *otherwise.*

Intuitively, J represents facts currently known to be true or false and $\Psi_J(I)$ contains all atomic facts whose truth can be derived *in one step* from the program P assuming that all facts in J hold and assuming that all positive facts in I are true. The Van Emden-Kowalski operator Ψ coincides with Ψ_J, where $J = <\emptyset, \emptyset>$. Observe, that operators Ψ_J are asymmetric in the sense that they *do not* treat negative and positive information symmetrically. In Section 6.7 devoted to the well-founded semantics we will introduce analogous, but completely symmetric operators.

Proposition 6.3 *[ABW88] For every interpretation J, the operator Ψ_J is monotone and it has the least fixed point given by $\Psi_J^{\uparrow\omega}$ (recall that ω stands for the first infinite ordinal).*

Intuitively, the least fixed point $\Psi_J^{\uparrow\omega}$ contains all positive (atomic) facts which can be derived from P knowing J. Now we give an iterated fixed point characterization of perfect models. Let $\{S_1, S_2, \ldots, S_\alpha, \ldots\}$, $\alpha < \lambda$, be a local stratification of a program P, i.e., a decomposition of the Herbrand base $\mathcal{H}$. For every $\beta \leq \lambda$ let

$$\mathcal{H}_\beta = \bigcup_{\alpha < \beta} S_\alpha.$$

Clearly,

$$\mathcal{H} = \mathcal{H}_\lambda.$$

Since the result of applying an operator Ψ_J to an arbitrary interpretation I is always a 2-valued interpretation $\Psi_J(I)$, we can identify interpretations $\Psi_J(I)$ with subsets of the Herbrand base. We construct the following (transfinite) sequence $\{I_\alpha : \alpha \leq \lambda\}$ of interpretations:

$$\begin{aligned} I_0 &= <\emptyset, \emptyset> \\ I_{\alpha+1} &= <\Psi^\omega_{I_\alpha}, \mathcal{H}_{\alpha+1} - \Psi^\omega_{I_\alpha}> \\ I_\delta &= \Sigma_F\{I_\alpha : \alpha < \delta\}, \end{aligned}$$

for limit δ. At any given step α, the next iteration $I_{\alpha+1}$ is obtained by:

- Taking the least fixed point $\Psi^{\omega}_{I_\alpha}$ of the operator Ψ_{I_α} as the set of positive facts. This is justified by the fact that the least fixed point $\Psi^{\omega}_{I_\alpha}$ contains those positive facts whose truth can be deduced from P assuming I_α.
- Taking the complement of $\Psi^{\omega}_{I_\alpha}$ in $\mathcal{H}_{\alpha+1}$ as the set of false facts. This is justified by the fact that the program is locally stratified and thus atoms from $\mathcal{H}_{\alpha+1}$, whose truth cannot be deduced at the level $\alpha + 1$ can be assumed to be false.

One can show that the sequence $\{I_\alpha\}$ is F-increasing and, clearly, the last interpretation in this sequence is I_λ. Observe again, that the above definition of iterations I_α does not treat negative and positive information symmetrically. In Section 6.7 devoted to the well-founded semantics we will give an analogous, but symmetric definition.

The following two theorems generalize results obtained in [ABW88, VG89b] from the class of stratified programs to the class of locally stratified programs. The approach presented here is slightly different from those given in [ABW88, VG89b].

Theorem 6.10 *(cf. [ABW88, VG89b]) The unique perfect model M_P of a locally stratified program coincides with I_λ. Moreover, M_P is itself the least fixed point of the operator Ψ_{M_P}.*

Thus perfect models of locally stratified programs can be viewed as iterated least fixed points of operators Ψ_J. Perfect models can be also described as *iterated least models* of the program.

Let us first denote by P_α the set of all (instantiated) clauses of P whose heads belong to $\mathcal{H}_\alpha$. Clearly, $P_\lambda = P$. It is easy to see that if in the above definition of the sequence I_α we replace the definition of $I_{\alpha+1}$ by:

$$I_{\alpha+1} =< \mathcal{H}_{\alpha+1} \cap \Psi^{\omega}_{I_\alpha}, \mathcal{H}_{\alpha+1} - \Psi^{\omega}_{I_\alpha} >$$

i.e., if we restrict true atoms to the elements of $\mathcal{H}_{\alpha+1}$ then we will still have $M_P = I_\lambda$. For the so modified sequence I_α the following result holds.

Theorem 6.11 *(cf. [ABW88, VG89b]) For every $\alpha \leq \lambda$, I_α is the least model of the program P_α, which extends all models I_β of programs P_β, for $\beta < \alpha$ (i.e., such that $I_\beta \preceq_F I_\alpha$).*

Thus M_P can be viewed as an iterated least model of P.

6.4 Extensions of the Perfect Model Semantics

The class of perfect models of locally stratified logic programs has many natural and desirable properties. However, the fact that it is restricted to the class of locally stratified programs is a significant drawback. Several researchers pointed out that there exist interesting and useful logic programs with natural intended semantics, which are not locally stratified [GL88, VGRS90].

Example 6.9 [GL88] Consider the program P given by:

$$\begin{aligned} p(1,2) &\leftarrow \\ q(X) &\leftarrow p(X,Y), \neg q(Y). \end{aligned}$$

After instantiating, P takes the form:

$$\begin{aligned} p(1,2) &\leftarrow && (6)\\ q(1) &\leftarrow p(1,2), \neg q(2) && (7)\\ q(1) &\leftarrow p(1,1), \neg q(1) && (8)\\ q(2) &\leftarrow p(2,2), \neg q(2) && (9)\\ q(2) &\leftarrow p(2,1), \neg q(1). && (10) \end{aligned}$$

This program is not locally stratified. Indeed, the priority relation $<$ between atoms is not a partial order, because $q(1) < q(2)$ and $q(2) < q(1)$. On the other hand, it seems clear that the intended semantics of P is well-defined and is characterized by the 2-valued model $M = \{p(1,2), q(1)\}$ of P. The same results would be produced by Prolog, which further confirms our intuition.

The cause of this peculiarity is fairly clear. Program P appears to be semantically equivalent to a locally stratified program P^* consisting only of clauses (6) and (7). The remaining clauses seem to be entirely irrelevant, because atoms $p(1,1)$, $p(2,1)$ and $p(2,2)$ can be assumed false in P. At the same time, they are the ones that destroy local stratifiability of P.

Three different extensions of the perfect model semantics have been proposed: the *stable model semantics* [GL88] (equivalent to the *default model semantics* [BF91]), the *weakly perfect model semantics* [PP88] and the *well-founded semantics* [VGRS90]. While the first two semantics are 2-valued and are defined only for restricted classes of programs, the well-founded semantics is 3-valued and is defined for *all* logic programs. Although the three semantics approach the problem from three different angles (see Section 7), it appears, that the well-founded semantics is the most adequate extension of the perfect model semantics. In the next section we discuss the weakly perfect model semantics. The remaining two semantics will be discussed in the following sections.

6.5 Weakly Perfect Model Semantics

As it was the case with locally stratified programs and perfect models, *the weakly perfect model semantics* is based on the decomposition of the program into *strata* and its semantics is based on the *iterated least model approach*, however, the decomposition is performed *dynamically* rather than *statically*. In the case of a single stratum the weakly perfect semantics is equivalent to the *least model semantics*. In this section all interpretations are assumed to be 2-valued.

The main idea behind the concept of the weakly perfect model semantics is to remove 'irrelevant' relations in the dependency graph G_P of a logic program

and to substitute *components* of the dependency graph for its *vertices* in the definitions of stratification and perfect models. First, we define the notion of a component.

Definition 6.11 *[PP88] Let* $\sim$ *be the equivalence relation between ground atoms of P defined as follows:*

$$A \sim B \equiv (A = B) \vee (A < B \wedge B < A).$$

We will call its equivalence classes components *of* G_P. *A component is* trivial *if it consists of a single element A, such that* $A \not< A$.

According to the above definition, two distinct ground atoms A and B are equivalent if they are related by *mutual negative recursion*, i.e., recursion passing through negative literals. Mutual negative recursion is the primary cause of difficulties with a proper definition of declarative semantics of logic programs. We now introduce an order relation $\prec$ between the *components* of the dependency graph G_P.

Definition 6.12 *[PP88] Let* C_1 *and* C_2 *be two components of* G_P. *We define:*

$$C_1 \prec C_2 \equiv C_1 \neq C_2 \wedge \exists A_1 \in C_1, \exists A_2 \in C_2 \ (A_1 < A_2).$$

We call a component C_1 maximal, *if there is no component* C_2 *such that* $C_1 \prec C_2$.

Clearly, the relation $\prec$ and therefore the maximal components of P are completely determined by the syntactic form of the program P. The order relation $\prec$ between the components of the dependency graph G_P corresponds to the dependency relation $<$ between its vertices, but, as opposed to $<$, it has the added advantage of *always being a partial order*. It easily follows from Proposition 6.2, that a logic program P is locally stratified if and only if all components of G_P are trivial and there is no infinite increasing sequence of components. For any logic program P we introduce the following definitions.

Definition 6.13 *[PP88] By the* bottom stratum S(P) *of P we mean the union of all maximal components of P:*

$$S(P) = \bigcup \{C : \quad C \text{ is a maximal component of } G_P\}.$$

By the bottom layer L(P) *of P we mean the corresponding subprogram of P:*

$$L(P) = \quad \text{the set of all clauses from } P, \text{ whose heads belong to } S(P).$$

Observe that, if the instantiated program is *infinite*, then it may not have any maximal components and thus its bottom stratum may be empty. For example, the bottom stratum of the program $p(X) \leftarrow \neg p(f(X))$ is empty.

Example 6.10 Consider the program P from Example 6.9. The dependency ordering is given by the following relations:

$$q(1) < q(2),\ q(2) < q(1),$$

$$q(1) \leq p(1,2),\ q(1) \leq p(1,1),\ q(2) \leq p(2,2),\ q(2) \leq p(2,1).$$

Program P has five components: $C_1 = \{q(1), q(2)\}, C_2 = \{p(1,2)\}, C_3 = \{p(1,1)\}, C_4 = \{p(2,2)\}$ and $C_5 = \{p(2,1)\}$. Clearly, $C_1 \prec C_k$, for any $2 \leq k \leq 5$. Consequently, the bottom stratum S(P) of P – defined as the union of maximal components of P – is given by:

$$S(P) = \{p(1,2), p(1,1), p(2,2), p(2,1)\},$$

and the bottom layer L(P) of P, i.e., the set of all clauses from P whose heads belong to S(P), is:

$$L(P) = \{p(1,2) \leftarrow\}.$$

Observe, that the bottom layer L(P) of the above program P has the least model

$$M = < \{p(1,2)\},\ \{p(1,1), p(2,2), p(2,1)\} >$$

i.e., the model in which $p(1,2)$ is true and $p(1,1),\ p(2,2),\ p(2,1)$ are false.

If the bottom layer L(P) of P has the least model M then we can use it to remove from P all 'irrelevant' clauses and literals. More generally, we will now introduce an operation of reduction, which reduces a given program P modulo its 3-valued interpretation I, by essentially applying the Davis-Putnam rule to P [CL73].

Definition 6.14 *[PP88] Let P be a logic program and let I be its interpretation. By a* reduction of P modulo I *we mean a new program* $\frac{P}{I}$ *obtained from P by performing the following two reductions:*

- *removing from P all clauses which contain a premise L such that* $\hat{I}(L) = 0$ *or whose head A satisfies* $I(A) = 1$*;*
- *removing from all the remaining clauses those premises L which satisfy* $\hat{I}(L) = 1$*.*

Finally, we also remove from the resulting program all non-unit clauses, whose heads already appear as unit clauses (facts) in the program. This step ensures that the set of atoms appearing in unit clauses, also called extensional *atoms, is disjoint from the set of atoms appearing in heads of non-unit clauses, also called* intensional *atoms.*

The so reduced program $\frac{P}{I}$ does not contain any literals which are true or false in I. In the Example 6.10 the reduced program $P' = \frac{P}{I}$ consists only of the clause:

$$q(1) \leftarrow \neg q(2).$$

Clearly, P' does not contain any literals from S(P). Observe that in the reduced program we got rid of all the 'irrelevant' clauses.

For 3-valued interpretations, $I_1 =< T_1, F_1 >$ and $I_2 =< T_2, F_2 >$, by their *F-union* $I_1 \cup_F I_2$ we denote the tuple $< T_1 \cup T_2, F_1 \cup F_2 >$. Clearly, $I_1 \cup_F I_2 = \Sigma_F\{I_1, I_2\}$. The idea behind the construction of weakly perfect models is as follows. Take any program $P = P_0$ and let $M_0 =< \emptyset, \emptyset >$. Let $P_1 = \frac{P}{M_0}$, find the least model M_1 of the bottom layer $L(P_1)$ of P_1 and reduce P modulo $M_0 \cup M_1$ obtaining a new program $P_2 = \frac{P}{M_0 \cup M_1}$. Find its bottom layer $L(P_2)$ and its least model M_2 and let $P_3 = \frac{P}{M_0 \cup M_1 \cup M_2}$. Continue the process until either the resulting k-th program P_k is empty, in which case $M_P = M_1 \cup \ldots \cup M_{k-1}$ is the weakly perfect model of P, or, otherwise, until either $S(P_k)$ is empty or $L(P_k)$ does not have a least model, in which case the weakly perfect model of P is undefined.

We now formalize the above approach, by giving a transfinite definition of a weakly perfect model M_P of a logic program P.

Definition 6.15 *[PP88] Suppose that P is a logic program and let $P_0 = P$, $M_0 =< \emptyset, \emptyset >$. Suppose that $\alpha > 0$ is a countable ordinal such that programs P_δ and interpretations M_δ have been already defined for all $\delta < \alpha$. Let*

$$N_\alpha = \Sigma_F\{M_\delta : 0 < \delta < \alpha\},$$

$$P_\alpha = \frac{P}{N_\alpha},\ S_\alpha = S(P_\alpha),\ L_\alpha = L(P_\alpha).$$

- *If the program P_α is empty, then the construction stops and $M_P = N_\alpha$ is the* weakly perfect model *of P. The ordinal α is called the* depth of P *and is denoted by $\lambda(P)$. For $0 < \alpha < \lambda(P)$, the set S_α is called the α-th* weak stratum *of P and the program L_α is called the α-th* layer *of P.*

- *Else, if the bottom stratum $S_\alpha = S(P_\alpha)$ of P_α is empty or if the least model of the bottom layer $L_\alpha = L(P_\alpha)$ of P_α does not exist, then the construction also stops and the weakly perfect model of P is undefined.*

- *Otherwise, the interpretation M_α is defined as the least model of the bottom layer $L_\alpha = L(P_\alpha)$ of P_α and the construction continues.*

In the process of constructing the strata S_α, some ground atoms may be eliminated by the reduction and not fall into any stratum. Such atoms should be added to an arbitrary stratum, e.g. the first, and assumed false in M_P.

It is easy to see that the construction always stops after countably many steps. A particularly important case of the above definition occurs when all the strata S_α consist only of trivial components or – equivalently – when all the program layers L_α are *positive* logic programs.

Definition 6.16 *[PP88] We say that a logic program P is* weakly stratified *if it has a weakly perfect model and if all of its strata S_α consist only of trivial components or – equivalently – when all of its layers L_α are positive logic programs. In this case, we call the set of program's strata $\{S_\alpha : 0 < \alpha < \lambda(P)\}$ the* weak stratification *of P.*

Remark 6.1 Observe, that since every positive logic program has the least model, a program P is weakly stratified if and only if whenever P_α is non-empty, $S_\alpha = S(P_\alpha)$ is also non-empty and consists only of trivial components.

Example 6.11 Consider the program P from Example 6.10. We obtain:

$$P_1 = P, \quad S_1 = S(P) = \{p(1,2), p(1,1), p(2,2), p(2,1)\},$$

$$L_1 = L(P) = \{p(1,2) \leftarrow\},$$

and therefore

$$M_1 =< \{p(1,2)\}, \{p(1,1), p(2,2), p(2,1)\} > .$$

Consequently, $P_2 = \frac{P_1}{M_1} = \{q(1) \leftarrow \neg q(2)\}$, $S_2 = S(P_2) = \{q(2)\}$ is the union of maximal components of P_2 and $L_2 = L(P_2) = \emptyset$ is the set of clauses from P_2 whose heads belong to S_2. Therefore, $M_2 =< \emptyset, \{q(2)\} >$. As a result,

$$P_3 = \frac{P_2}{M_1 \cup M_2} = \{q(1) \leftarrow\}, \quad S_3 = \{q(1)\}, \; L_3 = P_3 \text{ and } M_3 =< \{q(1)\}, \emptyset > .$$

Since $P_4 = \frac{P_3}{M_1 \cup M_2 \cup M_3} = \emptyset$, the construction is completed, P is weakly stratified, $\{S_1, S_2, S_3\}$ is its weak stratification and

$$M_P = M_1 \cup M_2 \cup M_3 =< \{p(1,2), q(1)\}, \{p(1,1), p(2,2), p(2,1), q(2)\} >$$

is its weakly perfect model.

The class of programs having weakly perfect models is much broader than the class of weakly stratified programs.

Example 6.12 Let P consist of clauses:

$$p \leftarrow q, \; q \leftarrow \neg p.$$

Then P has a single component and therefore its weakly perfect model is the least model of P, namely $M_P =< \{p\}, \{q\} >$ (see Corollary 6.14). Clearly, P is not weakly stratified. See [PP88, Example 3.4] for a discussion of this example.

Example 6.13 Let P be as follows:

$$\begin{aligned} p &\leftarrow q, \neg r \\ q &\leftarrow r, \neg p \\ r &\leftarrow p, \neg q \end{aligned}$$

The definition of propositions p, q and r is mutually circular, P has a single component and therefore its weakly perfect model is the least model of P, which is empty. Naturally, P is not weakly stratified.

The class of weakly stratified programs extends the class of (locally) stratified programs.

Theorem 6.12 *[PP88] Every locally stratified program is weakly stratified. Moreover, for locally stratified programs, the notions of a perfect model and a weakly perfect model coincide.*

Therefore, the weakly perfect model semantics *extends the perfect model* semantics from the class of locally stratified programs to a broader class of logic programs. Weakly perfect models also share the property of minimality with perfect models.

Theorem 6.13 *[PP88] Every weakly perfect model is minimal.*

It can be easily seen that the weakly perfect model semantics is based on the principle of *iterated (2-valued) least model semantics*. In particular, for programs with a single stratum the weakly perfect model semantics coincides with the least model semantics.

Corollary 6.14 *If a logic program P consists of a single stratum (in particular, if it consists of a single component), then for a model M of P:*

$$M \text{ is weakly perfect} \equiv M \text{ is perfect} \equiv M \text{ is the least model of } P.$$

The advantages of the weakly perfect model semantics include the facts that it extends the perfect model semantics, is patterned after the stratified approach and is based on the iterated least model approach. It is also closely related to the circumscriptive approach in non-monotonic reasoning (see Section 7). However, the weakly perfect model semantics also has some important drawbacks, namely, it is defined only for a restricted class of logic programs.

Example 6.14 [VGRS90] Let P be as follows:

$$\begin{aligned} p &\leftarrow q, \neg r, \neg s \\ q &\leftarrow r, \neg p \\ r &\leftarrow p, \neg q \\ s &\leftarrow \neg p, \neg q, \neg r. \end{aligned}$$

If we ignore the premise $\neg s$ in the first clause, then the first three clauses define p,q and r in a mutually circular fashion (see Example 6.13) and therefore we are compelled to assume the falsity of p, q and r and thus the intended semantics should imply s. The presence of the negative premise $\neg s$ does not seem to modify this conclusion. Unfortunatelly, it is easy to see that P consists of a single component and does not have the least model. Therefore, instead of producing the expected model $< \{s\}, \{p, q, r\} >$, the weakly perfect model semantics is undefined.

6.6 Stable Model Semantics

Stable models have been introduced in [GL88] by means of a fixed point definition, which uses a natural transformation of logic programs. We first introduce the *Gelfond-Lifschitz* operator Γ acting on *2-valued* interpretations of a given program P. All interpretations and models in this section are 2-valued.

Definition 6.17 (The Gelfond-Lifschitz Operator) *[GL88] Let P be a logic program and let I be its 2-valued interpretation. By a* GL-transformation of P modulo I *we mean a new program $\frac{P}{I}$ obtained from P by performing the following two reductions:*

- *removing from P all clauses which contain a negative premise $L = \neg A$ such that $\hat{I}(L) = 0$;*
- *removing from all the remaining clauses those negative premises $L = \neg A$ which satisfy $\hat{I}(L) = 1$.*

Since the resulting program $\frac{P}{I}$ does not contain any negative premises, it is positive and thus, by Theorem 6.4, it has a unique least model J. We define $\Gamma(I) = J$.

It turns out that fixed points of the Gelfond-Lifschitz operator Γ for a program P are always models of P.

Proposition 6.4 *[GL88] Fixed points of the Gelfond-Lifschitz operator Γ for a program P are minimal models of P.*

This result leads to the definition of the *stable model semantics.*

Definition 6.18 (Stable Model Semantics) *[GL88] A 2-valued interpretation I of a logic program P is called a* stable *model of P if $\Gamma(I) = I$. If a program P has a* unique *stable model M_P, then its* stable model semantics *is determined by this unique intended model M_P.*

Stable model semantics is closely related to the *autoepistemic* approach to non-monotonic reasoning (see [Gel87] and Section 7). The same semantics has been independently discovered in [BF91], where it is defined in terms of *default logic* and thus can be called the *default model semantics.*

Example 6.15 Consider the program P in Example 6.9 and let

$$M =< \{p(1,2), q(1)\}, \{p(1,1), p(2,2), p(2,1), q(2)\} >$$

be the unique weakly perfect model of P (see Example 6.11). The application of GL-transformation to P modulo M results in the following positive program $\frac{P}{M}$:

$$p(1,2) \leftarrow \tag{11}$$
$$q(1) \leftarrow p(1,2) \tag{12}$$
$$q(2) \leftarrow p(2,2) \tag{13}$$

whose least model is M itself. Therefore, $\Gamma(M) = M$, i.e., M is a fixed point of the operator Γ, and thus M is a stable model of P. It is easy to see that M is the only stable model of P. Consequently, the stable model semantics and the weakly perfect model semantics coincide in this case.

In general, a logic program may have one, none or many stable models. For example, the program $p \leftarrow \neg p$ does not have any stable models, while the program $p \leftarrow \neg q,\ q \leftarrow \neg p$ has two such models. However, in case of weakly stratified programs stable models are always unique.

Theorem 6.15 *[PP88] Every weakly stratified logic program P has a unique stable model M_P, which coincides with the unique weakly perfect model of P.*

In particular, the stable model semantics extends the perfect model semantics.

Corollary 6.16 *[GL88] Every locally stratified program P has a unique stable model which coincides with the unique perfect model of P.*

In general, beyond the class of weakly stratified programs, the stable model semantics and the weakly perfect model semantics are different.

Example 6.16 The program P in Example 6.12 has a weakly perfect model but it is easily seen not to have any stable models.

On the other hand, the program in Example 6.14 has the expected (intended) unique stable model $M_P =< \{s\}, \{p, q, r\} >$, but its weakly perfect model is undefined.

The advantages of the stable model semantics include the fact that it extends the perfect model semantics, has an elegant and simple definition and is closely related to autoepistemic and default approaches to non-monotonic reasoning (see Section 7). However, the stable model semantics also has some important drawbacks. First of all, it is defined only for a restricted class of programs and, secondly, it does not always seem to lead to the expected (intended) semantics.

Example 6.17 [[VGRS90]] Let P be given by:

$$\begin{aligned} b &\leftarrow \neg a \\ a &\leftarrow \neg b \\ p &\leftarrow \neg p \\ p &\leftarrow \neg a. \end{aligned}$$

Although its weakly perfect model is undefined, this program has a unique stable model $M = \{p, b\}$. However, the unique stable model M seems unintuitive [VGRS90] in view of the fact that, in 2-valued logic, p is a consequence of the third clause and therefore the last clause can be considered meaningless. The first two clauses, however, do not seem to have any reasonable (2-valued) intended semantics.

Moreover, it is easy to see that it is impossible to derive b from P using any Horn-resolution procedure. This is because any Horn-resolution procedure beginning with the goal $\leftarrow b$ will reach only the first two clauses of P, from which b cannot be derived.

As we will see in the next Section, the well-founded model semantics, which can be viewed as *3-valued stable model semantics* seems to avoid the difficulties encountered with the weakly perfect model semantics and the stable model semantics.

6.7 Well Founded Model Semantics

The well-founded semantics has been introduced in [VGRS90] and it seems to be the most adequate extension of the perfect model semantics to the class of *all* logic programs, avoiding various drawbacks of the other proposed approaches. Well-founded semantics also has been shown to be equivalent to suitable forms of 3-valued formalizations of all four major non-monotonic formalisms [Prz89d] (see Section 7).

One of the important features of well-founded models, and a strong indication of their naturality, is the fact that they can be described in many different, but equivalent, ways (see [VGRS90, Prz89a, Prz89d, VG89a, Bry89]). In this paper we use the (iterated) least fixed point approach proposed in [Prz89a], which seems to be a natural extension of least fixed point definitions of least models and perfect models and is also closely related to Fitting's extension of Clark's semantics. As opposed to the original definition proposed in [VGRS90], the iterated fixed point definition given here is constructive. In the second part of this section we will show that the well-founded semantics can be also viewed as *3-valued stable model semantics.*

First, for any interpretation J of a program P, we introduce the operator $\Theta_J : \mathcal{I} \rightarrow \mathcal{I}$ on the set $\mathcal{I}$ of all 3-valued interpretations of P, ordered by the standard ordering $\preceq$. The operator can be viewed as a cross between the Fitting

operator Φ (Section 6.1) and Generalized Van Emden-Kowalski operators Ψ_J (see Section 6.3).

Definition 6.19 *[Prz89a] Suppose that P is a logic program and J is its interpretation. The operator $\Theta_J : \mathcal{I} \rightarrow \mathcal{I}$ on the set $\mathcal{I}$ of all 3-valued interpretations of P is defined as follows. If $I \in \mathcal{I}$ is an interpretation of P and A is a ground atom then $\Theta_J(I)$ is an interpretation given by:*

(i) $\Theta_J(I)(A) = 1$ *if there is a clause $A \leftarrow L_1, \ldots, L_n$ in P such that, for all $i \leq n$, either $\hat{J}(L_i) = 1$ or L_i is positive and $I(L_i) = 1$;*

(ii) $\Theta_J(I)(A) = 0$ *if for every clause $A \leftarrow L_1, \ldots, L_n$ in P there is an $i \leq n$ such that either $\hat{J}(L_i) = 0$ or L_i is positive and $I(L_i) = 0$;*

(iii) $\Theta_J(I)(A) = \frac{1}{2}$ *, otherwise.*

Intuitively, the interpretation J represents facts *currently known* to be true or false. The *true* facts in $\Theta_J(I)$ consist of those atoms which can be derived *in one step* from the program P assuming that *all* facts in J hold and that all *positive* facts in I are true. The *false* facts in $\Theta_J(I)$ consist of those atoms whose falsity can be deduced *in one step* (using the closed world assumption) from the program P assuming that *all* facts in J hold and that all *negative* facts in I are true. Observe, that, as opposed to Van Emden-Kowalski operators Ψ_J, the operators Θ_J are completely *symmetric* in the sense that they treat negative and positive information symmetrically (dually).

Theorem 6.17 *[Prz89a] For every interpretation J, the operator Θ_J is monotone and it has a unique least fixed point*[11] *given by $\Theta_J^{\uparrow\omega}$.*

We will denote this least fixed point of Θ_J by $\Omega(J)$, i.e.

$$\Omega(J) = \Theta_J^{\uparrow\omega}.$$

Clearly, Ω constitutes an operator on the set of all interpretations of P.

Observe that, although the operators Θ_J resemble the Fitting operator Φ, they do not coincide with it. Moreover, the above Theorem is very different from Theorem 6.2. This is a consequence of the fact that the operators Θ_J are defined on the set of all interpretations ordered by the standard ordering $\preceq$ and not by the F-ordering $\preceq_F$ and thus the iterations begin from the smallest interpretation $I_0 =< \emptyset, \mathcal{H} >$ and not from the F-smallest interpretation $< \emptyset, \emptyset >$. As a result, as opposed to Φ, least points of operators Θ_J can be always obtained after only ω steps, where ω is the first infinite ordinal.

Intuitively, the least fixed point $\Omega(J) = \Theta_J^{\uparrow\omega}$ of Θ_J contains all facts, whose truth or falsity can be deduced (using the closed world assumption) from P knowing J. The operator Ω turns out to have a unique F-least fixed point.

[11] Recall that ω stands for the first infinite ordinal and that the iteration begins from the smallest interpretation $I_0 =< \emptyset, \mathcal{H} >$.

Theorem 6.18 *[Prz89a] The operator Ω always has a unique F-least fixed point M_P, i.e. the F-least interpretation M_P such that*

$$\Omega(M_P) = M_P.$$

Moreover, all fixed points of Ω are minimal models of P.

As we will see below, the F-least fixed point of Ω can be simply obtained as a suitable iteration $\Omega^{\uparrow\lambda}$ of Ω. First, we give a definition of well-founded models.

Definition 6.20 *[Prz89a] We call the unique F-least fixed point M_P of Ω the* well founded model *of P.*

Since the well-founded model of P is defined as the F-least fixed point of the operator Ω, which is itself defined by means of least fixed points of Θ, well founded models can be viewed as *iterated least fixed points* of the operator Θ. Although our definition of well-founded models is different from the original definition given in [VGRS90], the two notions are equivalent.

Theorem 6.19 *[Prz89a] Well founded models introduced above coincide with well-founded models originally defined in [VGRS90].*

Now we can introduce the well-founded semantics of logic programs.

Definition 6.21 (Well-Founded Semantics) *[VGRS90] The well founded semantics of a logic program is determined by the unique well-founded model M_P.*

In order to obtain a *constructive* definition of the well-founded model M_P of a given program P, we define the following (transfinite) sequence $\{I_\alpha\}$ of interpretations of P:

$$\begin{aligned} I_0 &= <\emptyset, \emptyset> \\ I_{\alpha+1} &= \Omega(I_\alpha) = \Theta_{I_\alpha}^{\uparrow\omega} \\ I_\delta &= \Sigma_F\{I_\alpha : \alpha < \delta\}, \end{aligned}$$

for limit δ. Clearly, for any α, I_α coincides with $\Omega^{\uparrow\alpha}$, where the F-ordering of interpretations is used to generate consecutive iterations, i.e., we have:

$$I_\alpha = \Omega^{\uparrow\alpha}.$$

At any given step α, the next iteration $I_{\alpha+1}$ is obtained as the least fixed point $\Omega(I_\alpha) = \Theta_{I_\alpha}^{\omega}$ of the operator Θ_{I_α}. This is justified by the fact that, as we observed before, the least fixed point $\Omega(I_\alpha) = \Theta_{I_\alpha}^{\omega}$ contains all facts, whose truth or falsity can be deduced from P knowing I_α.

One can show that the sequence $\{I_\alpha\}$ is well-defined and F-increasing and therefore, since all interpretations are countable, there must exist the smallest λ, such that I_λ is a fixed point , i.e., such that:

$$I_{\lambda+1} = \Omega(I_\lambda) = I_\lambda$$

i.e., I_λ is a fixed point of the the operator Ω. We call $\lambda = \lambda(P)$ the depth of the program P. It turns out that I_λ is in fact the F-least fixed point of Ω and thus it coincides with the well-founded model M_P of P.

Theorem 6.20 *[Prz89a] The interpretation $I_\lambda = \Omega^{\uparrow\lambda}$ is the F-least fixed point of the operator Ω and thus it coincides with the well-founded model M_P of P:*

$$M_P = I_\lambda = \Omega^{\uparrow\lambda}.$$

Observe, that the above description of well-founded models is very similar to the iterated fixed point definition of perfect models given in Section 6.3, but it treats negative and positive information completely symmetrically (dually) and does not require the advance notion of (local) stratification.

Example 6.18 Consider the program P from Example 6.9 given by:

$$\begin{array}{rcl} p(1,2) & \leftarrow & \\ q(1) & \leftarrow & p(1,2), \neg q(2) \\ q(1) & \leftarrow & p(1,1), \neg q(1) \\ q(2) & \leftarrow & p(2,2), \neg q(2) \\ q(2) & \leftarrow & p(2,1), \neg q(1). \end{array}$$

We have $I_0 =< \emptyset, \emptyset >$ and $I_1 = \Omega(I_1) = \Theta_{I_0}^{\uparrow\omega}$. Since $\Theta_{I_0}^{\uparrow 0} =< \emptyset, \mathcal{H} >$ it follows from the definition of Θ that:

$$\Theta_{I_0}^{\uparrow 1} = \Theta_{I_0}(< \emptyset, \mathcal{H} >) =< \{p(1,2)\}, \{p(2,1), p(1,1), p(2,2), q(1), q(2)\} > .$$

Similarly,

$$\Theta_{I_0}^{\uparrow 2} = \Theta_{I_0}(\Theta_{I_0}^{\uparrow 1}) =< \{p(1,2)\}, \{p(2,1), p(1,1), p(2,2), q(2)\} >$$

and it is easy to see that $\Theta_{I_0}^{\uparrow 2}$ is a fixed point of Θ_{I_0}, i.e.,

$$\Theta_{I_0}^{\uparrow 2} = \Theta_{I_0}^{\uparrow\omega} = I_1.$$

Now, $I_2 = \Omega(I_1) = \Theta_{I_1}^{\uparrow\omega}$ and since $\Theta_{I_1}^{\uparrow 0} =< \emptyset, \mathcal{H} >$ it follows from the definition of Θ that:

$$\Theta_{I_1}^{\uparrow 1} = \Theta_{I_1}(< \emptyset, \mathcal{H} >) =< \{p(1,2), q(1)\}, \{p(2,1), p(1,1), p(2,2), q(2)\} >$$

and it is easy to see that $\Theta_{I_1}^{\uparrow 1}$ is a fixed point of Θ_{I_1}, i.e.,

$$\Theta_{I_1}^{\uparrow 1} = \Theta_{I_1}^{\uparrow \omega} = I_2.$$

Moreover, it is clear that the construction stops here because $I_3 = \Omega(I_2) = I_2$ and the well founded model of P is 2-valued and given by:

$$M_P = I_2 =< \{p(1,2), q(1)\}, \{p(2,1), p(1,1), p(2,2), q(2)\} > .$$

The well-founded model coincides therefore with the weakly perfect model and the unique stable model of P. Moreover, the depth of the program is $\lambda(P) = 2$.

A particularly important class of logic programs consists of those programs which have a *2-valued* well-founded model.

Definition 6.22 *[Prz89a] We call a logic program P* saturated *if its well-founded model M_P is 2-valued.*

For example the above discussed program is saturated. For saturated logic programs, the well-founded and the stable model semantics coincide.

Theorem 6.21 *[VGRS90] If a program has a 2-valued well-founded model M_P then it also has a unique stable model and the two models coincide.*

For weakly stratified programs, all three extensions of the perfect model semantics coincide.

Theorem 6.22 *[PP88] For weakly stratified logic programs, well-founded models coincide with weakly perfect models and with unique stable models.*

In particular, the well-founded model semantics extends the perfect model semantics.

Corollary 6.23 *[VGRS90] The well-founded model of a locally stratified program coincides with its perfect model.*

In general, the well-founded semantics, the stable model semantics and the weakly perfect model semantics are different. The fact that the well-founded semantics is different from the weakly perfect semantics follows immediately from Theorem 6.21 and from the examples illustrating the differences between the stable model semantics and the weakly perfect model semantics given in the previous section. Moreover, the program $p \leftarrow \neg p$ does not have any stable models, but its well-founded model is $< \emptyset, \emptyset >$. There exist also programs, which admit unique (2-valued) stable models that do not coincide with well-founded models. For example, the well-founded model of the program presented in Example 6.17 is $< \emptyset, \emptyset >$, which seems to be more intuitive than its stable model $M =< \{p, b\}, \{a\} >$.

In [Prz89a] the iterated fixed point definition of well founded models is used to introduce the so called *dynamic stratification* of an *arbitrary* logic program P, with properties analogous to local stratification. Using dynamic stratification, [Prz89a] showed that the well-founded model M_P can also be viewed as an *iterated least model* of a program and that well-founded models can be defined by means of a suitable *preference relation* between atoms, in a manner analogous to the definition of perfect models.

6.7.1 Well-Founded Semantics Coincides With 3-Valued Stable Semantics

In this section we give a definition of *3-valued stable models* which is exactly analogous to the definition of 2-valued stable models given in the previous section. Every 2-valued stable model is a 3-valued stable model, so our definition extends the notion of stable models. We show that the well-founded model M_P of a program P is the *F-least stable model* of P. In particular, every logic program has at least one 3-valued stable model, namely M_P.

As we mentioned before, models of a program can be thought of as *possible worlds* representing possible states of our knowledge. Since, at any given moment, our knowledge about the world is likely to be incomplete, we need the ability to describe possible worlds in which some facts are neither true nor false and thus their status is unknown. This explains the need to use 3-valued logic or 3-valued possible worlds to describe our knowledge.

Before defining 3-valued stable models we need to expand our language by adding to it the proposition $\mathbf{u}$ denoting the property of being *unknown* or *undefined*. We assume that every interpretation I satisfies $I(\mathbf{u}) = \frac{1}{2}$ and thus $\hat{I}(\neg\mathbf{u}) = \frac{1}{2}$. We can therefore always replace $\neg\mathbf{u}$ by $\mathbf{u}$. By a *non-negative* program we mean a program, whose premises are either positive atoms or $\mathbf{u}$. First, we need the following generalization of the Kowalski-Van Emden Theorem 6.4:

Theorem 6.24 *Every non-negative logic program P has a unique least 3-valued model.*

We now extend the Gelfond-Lifschitz operator Γ to a 3-valued operator Γ^* as follows:

Definition 6.23 *Let P be a logic program and let I be its 3-valued interpretation. By the* extended GL-transformation of P modulo I *we mean a new program* $\frac{P}{I}$ *obtained from P by performing the following three operations:*

- *Removing from P all clauses which contain a negative premise $L = \neg A$ such that $\hat{I}(L) = 0$;*
- *Replacing in all the remaining clauses those negative premises $L = \neg A$ which satisfy $\hat{I}(L) = \frac{1}{2}$ by $\mathbf{u}$;*

- *Removing from all the remaining clauses those negative premises $L = \neg A$ which satisfy $\hat{I}(L) = 1$.*

Since the resulting program $\frac{P}{I}$ is non-negative thus, by Theorem 6.24, it has a unique least 3-valued model J. We define $\Gamma^*(I) = J$.

Fixed points of the operator Γ^* for a program P are defined as 3-valued stable models.

Definition 6.24 *A 3-valued interpretation I of a logic program P is called a* 3-valued stable *model of P if $\Gamma^*(I) = I$. The* 3-valued stable model semantics *of a program P is determined by the set $MOD(P)$ of all 3-valued stable models of P.*

It is easy to see that the above definition is a strict extension of the standard definition of stable models. In fact, standard stable models coincide with 2-valued stable models introduced above.

Example 6.19 Suppose that P is:

$$\begin{array}{lcl} c & \leftarrow & \neg d \\ a & \leftarrow & \neg b \\ b & \leftarrow & \neg a \end{array}$$

and let $M =< \{c\}, \{d\} >$. Then the transformed program is:

$$\begin{array}{lcl} c & \leftarrow & \\ a & \leftarrow & \mathbf{u} \\ b & \leftarrow & \mathbf{u} \end{array}$$

and its least model $\Gamma^*(M)$ coincides with M which shows that M is a 3-valued stable model of P.

Well-founded models are F-least stable models.

Theorem 6.25 *All stable models are minimal. The well-founded model M_P of a program P is the F-least 3-valued stable model of P. Consequently, the well-founded semantics coincides with the 3-valued stable model semantics.*

We can say, borrowing from Horty and Thomasson's inheritance network terminology, that the well founded model is the most *skeptical* 3-valued stable model or possible world for P. For example, if P is given by $a \leftarrow \neg b$, $b \leftarrow \neg a$, then P has three stable models. One, in which a is true and b is false, the second, exactly opposite and the third in which both a and b are undefined. The last model, the most 'skeptical' one, is the well-founded model of P.

Corollary 6.26 *Every logic program has at least one 3-valued stable model, namely, the well-founded model M_P.*

Recall, that not every logic program has 2-valued stable models. Also, observe that although the characterization of well founded models as F-least 3-valued stable models is mathematically elegant it does not provide any constructive way of building such models.

6.8 Semantics Based on Non-Herbrand Models

Throughout the paper, with the exception of Clark's semantics and its 3-valued extension due to Kunen, we restricted ourselves to *Herbrand* interpretations and models. This approach is very convenient, in most cases leads to semantics based on *one* intended Herbrand model and is often quite suitable for *deductive database* applications. However, from the point of view of *logic programming*, the Herbrand approach has an important drawback, which was called the *universal query problem* in [Prz89c].

Suppose that our program P consists of a trivial clause $p(a)$. The program is positive and has only one (2-valued) Herbrand model $M_P = \{p(a)\}$. Therefore all model-theoretic semantics of P based on Herbrand models coincide and are determined by the model M_P. Consequently, all such semantics imply $\forall X\ p(X)$, because

$$M_P \models \forall X\ p(X).$$

In addition to not being very intuitive, this conclusion causes at least two negative consequences:

- Since $\forall X\ p(X)$ is a positive formula, not implied by P itself, all semantics based on Herbrand models of P violate the principle that *no new positive information should be introduced by the semantics of positive programs*, which – as it was explained in [Prz89c] – seems to be a natural and important requirement in logic programming.

- They also seem to *a priori* prevent standard unification-based computational mechanisms, typically used in logic programming, from being complete with respect to this semantics.

Indeed, when we ask the query $p(X)$ in logic programming, we not only want to have an answer to the question 'is there an X for which p(X) holds?', but, in fact, we are interested in obtaining *all* most general bindings (or substitutions) θ for which our semantics implies $\forall X\ p(X)\theta$. Therefore, in this case, if we ask $\leftarrow p(X)$, we should expect simply the answer 'yes' indicating that p(X) is satisfied for all X's or – in other words – signifying, that the empty substitution is a correct answer substitution. Unfortunately, standard unification-based computational mechanisms will be only capable of returning the special case substitution $\theta = \{X|a\}$.

It is sometimes argued that logic programming should only be concerned with Herbrand models rather than with general models of P. This conclusion is motivated by the belief that the role of logic programming is to answer existential queries and by the well-known fact that an existential formula F is derivable from a given (universal) theory T if and only if it is satisfied in all Herbrand models of T. This argument is only partially correct. In reality, *logic programming is not only concerned with answering existential queries, but it is primarily concerned with providing 'most general' bindings (substitutions) for the answers.* For example, if our program is

$$\begin{aligned} &parent(X, father(X)) \\ &parent(X, mother(X)) \\ &grand\_parent(X, Y) \quad \leftarrow \quad parent(X, Z), parent(Z, Y) \end{aligned}$$

and we ask $\leftarrow grand\_parent(X, Y)$, then we expect to obtain answers:

$$Y = mother(father(X)),\ Y = mother(mother(X))$$

etc., signifying that

$$\begin{aligned} &\forall X\ grand\_parent(X, mother(father(X))) \\ &\forall X\ grand\_parent(X, mother(mother(X))), \ldots \end{aligned}$$

In other words, we expect to obtain 'most general' substitutions for which the given query holds and, as a result, we are in fact interested in answers to universal queries, like *'Is it true that for every X grandparent(X,mother(father(X)))?'*, to which general models and Herbrand models often provide different answers, as it was illustrated above.

There are two natural solutions to the universal query problem:

1. One can stick to Herbrand models of the program, but in addition:

   - Either *extend the language* of the program by asserting the existence of *infinitely* many function symbols (or constants) (see e.g. [Kun87]);
   - Or *extend the language* by asserting the existence of one or more 'dummy' functions (see e.g. [VGRS90]), which exist in the language, but are not used in the program.

   From the semantic point of view these two approaches are essentially equivalent, but they also share a common problem, namely in some cases they may not be very natural. The reason is that one may not wish to automatically assume the *existence of objects* that are not mentioned *explicitly* in the program. Such an assumption can be called an *infinite domain assumption* and can be viewed as being in some sense *opposite* to the closed world assumption. In its presence, if we know only that $p(a)$ holds, then we are forced to conclude that there are many x's for which $p(x)$ is false, which may not always be desirable.

2. Another approach (see [Prz89c]) is to *extend the definitions of intended models* to include *non-Herbrand* models, thus leading to the definitions of non-Herbrand perfect models (resp. non-Herbrand stable models, non-Herbrand well founded models, etc.). One then defines the corresponding semantics to be determined by the set $MOD(P)$ of all, *Herbrand and non-Herbrand*, perfect models (resp. stable models, well founded models, etc.). Using this approach and knowing only that $p(a)$ holds, the answer to the query 'Does there exist an x for which p(x) is false?' is undefined, which in some contexts may seem more reasonable.

   The extension of the definition of intended models, so that they include non-Herbrand models, is usually quite straightforward. For the perfect model semantics it has been done in [Prz89c] and for the other semantics it can be done in an analogous way.

However, when using non-Herbrand models in the context of logic programming, one has to additionally assume that they satisfy to so called *Clark's Equational Theory* (CET) [Kun87]:

CET1. $X = X$;

CET2. $X = Y \Rightarrow Y = X$;

CET3. $X = Y \wedge Y = Z \Rightarrow X = Z$;

CET4. $X_1 = Y_1 \wedge ... \wedge X_m = Y_m \Rightarrow f(X_1, ..., X_m) = f(Y_1, ..., Y_m)$, for any function f;

CET5. $X_1 = Y_1 \wedge ... \wedge X_m = Y_m \Rightarrow (p(X_1, ..., X_m) \Rightarrow p(Y_1, ..., Y_m))$, for predicate p;

CET6. $f(X_1, ..., X_m) \neq g(Y_1, ..., Y_n)$, for any two different function symbols f and g;

CET7. $f(X_1, ..., X_m) = f(Y_1, ..., Y_m) \Rightarrow X_1 = Y_1 \wedge ... \wedge X_m = Y_m$, for any function f;

CET8. $t[X] \neq X$, for any term $t[X]$ different from X, but containing X.

The first five axioms describe the usual *equality axioms* and the remaining three axioms are called *unique names axioms* or *freeness axioms*. The significance of these axioms to logic programming is widely recognized [Llo84, Kun87].

The equality axioms (CET1) – (CET5) ensure that we can always assume that the *equality predicate = is interpreted as identity* in all models. Consequently, in order to satisfy the CET axioms, we just have to restrict ourselves to those models in which the equality predicate – when interpreted as identity – satisfies the unique names axioms (CET6) – (CET8).

For more information about the relationship between approaches based on Herbrand and on non-Herbrand models see [GPP88, Prz89c].

7 Relationship to Non-Monotonic Formalisms

Non-monotonic reasoning, logic programming and deductive databases are areas of crucial and growing significance to Artificial Intelligence and to the whole field of computer science. It is therefore important to achieve a better understanding of the relationship existing between these three fields.

There is no doubt that the three areas are related. Logic programming and database systems implement negation using various non-monotonic negation operators, such as the negation as failure mechanism of Prolog. The non-monotonic character of those operators closely relates logic programming and deductive databases to non-monotonic reasoning. Conversely, because of the non-monotonic character of such procedural operators, they can often be used to implement other non-monotonic formalisms [Rei86], thus opening the possibility for using logic programming and deductive databases as inference engines for non-monotonic reasoning.

In spite of the close relationship between non-monotonic reasoning, on the one hand, and logic programming and deductive databases, on the other, in the past these research areas have been developing largely independently of one another and the exact nature of their relationship has not been closely investigated or understood. One possible explanation of this phenomenon is the fact that, traditionally (see [Llo84]), the declarative semantics of logic programs has been based on the Clark predicate completion (see Section 6.1). Clark's formalism, although very elegant and natural, is not sufficiently general to be applied beyond the realm of logic programming and therefore does not play a major role in formalizing general non-monotonic reasoning in AI.

The situation has changed significantly with the introduction of stratified logic programs and the perfect model semantics (see Section 6.3). For locally stratified logic programs, the perfect model semantics has been shown (see [Prz88b] for an overview) to be *equivalent* to natural forms of *all four* major formalizations of non-monotonic reasoning in AI:

- McCarthy's circumscription [Prz89c, Lif88];
- Reiter's default theory [BF87];
- Moore's autoepistemic logic [Gel87];
- Reiter's CWA [GPP89].

These results shed a new light on the semantics of logic programs and deductive databases and established a closer link between non-monotonic reasoning, logic programming and deductive databases.

As we know, three essentially different extensions of the perfect model semantics have been proposed:

- *Stable model semantics* (or, equivalently, *default model semantics*) (see Section 6.6) has been shown to be equivalent to natural forms of autoepistemic logic and default theory [Gel87, BF91].
- *Weakly perfect model semantics* (see Section 6.5) has been shown to be equivalent to natural forms of circumscription and CWA [PP88].
- *Well-founded model semantics* (see Section 6.7): originally, its relationship to non-monotonic formalisms was unclear.

In view of this situation, it initially appeared that it will not be possible to extend the result stating the equivalence of the perfect model semantics to suitable forms of all four major non-monotonic formalisms to much broader classes of programs. Nevertheless, Przymusinski has shown in [Prz89d] that the well-founded semantics is in fact also *equivalent* to natural forms of *all four* major formalizations of non-monotonic reasoning. However, in order to achieve this equivalence, *3-valued extensions of non-monotonic formalisms* had to be introduced, which is natural in view of the fact that the well-founded semantics is also 3-valued.

This above mentioned results will likely contribute to a better understanding of relations existing between various formalizations of non-monotonic reasoning and, hopefully, to the eventual discovery of deeper underlying principles of non-monotonic reasoning. They also pave the way for using *efficient computation methods*, developed for logic programs and deductive databases, as inference engines for non-monotonic reasoning.

For more information about the relationship between non-monotonic reasoning, logic programming and deductive databases, the reader is referred to [Prz88b, Prz89b].

8 Conclusion

In the paper we have described various proposed semantics for deductive databases and logic programs and discussed their mutual relations. We have particularly emphasized recent research work in this area which appears to be very significant and lead to a change of perspective. As a result a fairly clear picture seems to emerge:

- Fitting's and Kunen's semantics both appear to be suitable 3-valued *extensions of Clark's predicate completion semantics* to the class of all logic programs. The differences between the two approaches simply reflect the differences existing between semantics based exclusively on Herbrand models and those including non-Herbrand models (see Section 6.8). Consequently, Kunen's semantics may be preferred in the context of logic programming, while Fitting's semantics may be more suitable for deductive databases.

Both semantics inherit the drawbacks of Clark's semantics and do not closely relate to non-monotonic reasoning.

- There are two essentially different *2-valued model-theoretic* semantics of logic programs, both of which are closely related to non-monotonic formalisms. One of them is the *stable model semantics* [GL88], which coincides with the *default model semantics* [BF91] and is based on autoepistemic logic or default theory. The other is the *weakly perfect model semantics* [PP88], based on circumscription or CWA.

- There is a unique *3-valued model-theoretic* semantics, namely the *well-founded semantics* [VGRS90], which is equivalent to suitable forms of *all* 3-valued non-monotonic formalisms and appears to be the most adequate semantics for logic programs and deductive databases.

- The well-founded semantics is defined for all logic programs, whereas the 2-valued semantics are restricted to more narrow domains. All the three model-theoretic semantics extend the *perfect model semantics* of stratified programs and coincide in the class of *weakly stratified programs* [PP88].

Acknowledgments

The authors are grateful to Michael Gelfond, Vladimir Lifschitz and Allen Van Gelder for helpful discussions on the subject of this article.

References

[ABW88] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–142. Morgan Kaufmann, Los Altos, CA., 1988.

[BF87] N. Bidoit and C. Froidevaux. Minimalism subsumes default logic and circumscription in stratified logic programming. In *IEEE Symposium on Logic in Computer Science*, pages 89–97, Ithaca, New York, USA, June 1987.

[BF91] N. Bidoit and C. Froidevaux. General logical databases and programs: Default logic semantics and stratification. *Journal of Information and Computation*, pages 15–54, 1991.

[Bry89] F. Bry. Logic programming as constructivism: A formalization and its application to databases. In *Proceedings of the Symposium on Principles of Database Systems*, pages 34–50. ACM SIGACT-SIGMOD, 1989.

[BS85] G. Bossu and P. Siegel. Saturation, non-monotonic reasoning and the closed world assumption. *Journal of Artificial Intelligence*, 25:13–63, 1985.

[CH85] A. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 1:1–15, 1985.

[CKRP73] A. Colmerauer, H. Kanoui, P. Roussel, and R. Passero. Un systeme de communication homme-machine en francais. Research report, Groupe de Recherche en Intelligence Artificielle, Universite d'Aix-Marseille, 1973.

[CL73] C. Chang and R.C. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

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

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

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

[Gel87] M. Gelfond. On stratified autoepistemic theories. In *Proceedings AAAI-87*, pages 207–211, Los Altos, CA, 1987. American Association for Artificial Intelligence, Morgan Kaufmann.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth Logic Programming Symposium*, pages 1070–1080, Cambridge, Mass., 1988. Association for Logic Programming, MIT Press.

[GM78] H. Gallaire and J. Minker. *Logic and Data Bases*. Plenum Press, New York, 1978.

[GMN84] H. Gallaire, J. Minker, and J. Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16:153–185, 1984.

[GPP88] M. Gelfond, H. Przymusinska, and T. Przymusinski. Minimal model semantics vs. negation as failure: A comparison of semantics. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems*, pages 335–343. ACM SIGART, 1988.

[GPP89] M. Gelfond, H. Przymusinska, and T. Przymusinski. On the relationship between circumscription and negation as failure. *Journal of Artificial Intelligence*, 38:75–94, 1989.

[JLL83] J. Jaffar, J-L. Lassez, and J. Lloyd. Completeness of the negation as failure rule. In *Proceedings AAAI-83*, pages 500–506, Los Altos, CA, 1983. American Association for Artificial Intelligence, Morgan Kaufmann.

[Kle52] S.C. Kleene. *Introduction to Meta-mathematics.* Van Nostrand, Princeton, 1952.

[Kow74] R. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP-74*, pages 569–574, 1974.

[Kow79] R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424–436, 1979.

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

[Kun88] K. Kunen. Some remarks on the completed database. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth Logic Programming Symposium*, pages 978–992, Cambridge, Mass., 1988. Association for Logic Programming, MIT Press.

[Lif88] V. Lifschitz. On the declarative semantics of logic programs with negation. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 177–192. Morgan Kaufmann, Los Altos, CA., 1988.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, New York, N.Y., first edition, 1984.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, New York, N.Y., second edition, 1987.

[LT85] J.W. Lloyd and R.W. Topor. A basis for deductive database systems. *Journal of Logic Programming*, 2:93–109, 1985.

[LT86] J.W. Lloyd and R.W. Topor. A basis for deductive database systems II. *Journal of Logic Programming*, 3:55–67, 1986.

[McC80] J. McCarthy. Circumscription – a form of non-monotonic reasoning. *Journal of Artificial Intelligence*, 13:27–39, 1980.

[Min82] J. Minker. On indefinite data bases and the closed world assumption. In *Proc. 6-th Conference on Automated Deduction*, pages 292–308, New York, 1982. Springer Verlag.

[Min88a] J. Minker. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, Los Altos, CA., 1988.

[Min88b] J. Minker. Perspectives in deductive databases. *Journal of Logic Programming*, 5(1):33–60, 1988.

[MUG86] K. Morris, J.D. Ullman, and A. Van Gelder. Design overview of the NAIL! system. In *Proceedings of the Third International Conference on Logic Programming, London, July 1986*. Association for Logic Programming, Springer Verlag, 1986.

[Naq86] S.A. Naqvi. A logic for negation in database systems. In J. Minker, editor, *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming, Washington, D.C.*, pages 378–387, August 1986.

[PP88] H. Przymusinska and T. C. Przymusinski. Weakly perfect model semantics for logic programs. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth Logic Programming Symposium*, pages 1106–1122, Cambridge, Mass., 1988. Association for Logic Programming, MIT Press.

[Prz88a] T. C. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos, CA., 1988.

[Prz88b] T. C. Przymusinski. On the relationship between non-monotonic reasoning and logic programming. In *Proceedings AAAI-88*, pages 444–448, Los Altos, CA, 1988. American Association for Artificial Intelligence, Morgan Kaufmann. [The full version appeared in: T. C. Przymusinski. Non-Monotonic Reasoning vs. Logic Programming: A New Perspective. In *The Foundations of Artificial Intelligence. A Sourcebook*, D. Partridge and Y. Wilks, editors, Cambridge University Press, London, 1990, 49-71.].

[Prz89a] T. C. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *Proceedings of the Eighth Symposium on Principles of Database Systems*, pages 11–21. ACM SIGACT-SIGMOD, 1989.

[Prz89b] T. C. Przymusinski. Non-monotonic formalisms and logic programming. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Logic Programming Conference, Lisbon, Portugal*, pages 655–674, Cambridge, Mass., 1989. Association for Logic Programming, MIT Press.

[Prz89c] T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[Prz89d] T. C. Przymusinski. Three-valued non-monotonic formalisms and logic programming. In R. Brachman, H. Leveque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89), Toronto, Canada*, pages 341–348. Morgan Kaufmann, 1989.

[Rei78] R. Reiter. On closed-world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[Rei84] R. Reiter. Towards a logical reconstruction of relational database theory. In M. Brodie and J. Mylopoulos, editors, *On Conceptual Modelling*, pages 191–233. Springer Verlag, New York, 1984.

[Rei86] R. Reiter. Nonmonotonic reasoning. *Annual Reviews of Computer Science*, 1986.

[Rou75] P. Roussel. Prolog, manuel de reference et d'utilisation. Research report, Group d'Intelligence Artificielle, U.E.R. de Marseille, France, 1975.

[She84] J. Shepherdson. Negation as finite failure: A comparison of Clark's completed data bases and Reiter's closed world assumption. *Journal of Logic Programming*, 1:51–79, 1984.

[She88] J.C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, Los Altos, CA., 1988.

[Ull88] J.D. Ullman. *Database and Knowledge-Based Systems.* Computer Science Press, Rockville, Md., 1988.

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

[VG89a] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proceedings of the Symposium on Principles of Database Systems*, pages 1–10. ACM SIGACT-SIGMOD, 1989.

[VG89b] A. Van Gelder. Negation as failure using tight derivations for general logic programs. *Journal of Logic Programming*, 6(1):109–133, 1989. Preliminary versions appeared in *Third IEEE Symposium Logic Programming* (1986), and *Foundations of Deductive Databases and Logic Programming*, J. Minker, ed., Morgan Kaufmann, 1988.

[VGRS90] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 1990. (to appear). Preliminary abstract appeared in Seventh ACM Symposium on Principles of Database Systems, March 1988, pp. 221–230.

[Zan88] C. Zaniolo. Design and implementation of logic-based language for data intensive applications. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth Logic Programming Symposium*, pages 1666–1688, Cambridge, Mass., 1988. Association for Logic Programming, MIT Press.