Well-Founded and Stationary Models of Logic Programs

Teodor C. Przymusinski

Department of Computer Science
University of California
Riverside, CA 92521
(teodor@cs.ucr.edu)*

Table of Contents

1 Introduction 1

2 Clark's Predicate Completion Semantics and its Drawbacks . 2
2.1 The Negative Recursion Problem: Inconsistency 4
2.2 The Positive Recursion Problem: Insufficient Expressibility . . . 5

3 Eliminating Drawbacks of Clark's Semantics 7
3.1 Partial Clark's Predicate Completion Semantics 7
3.2 Least Models 8
3.3 Perfect Models 9
3.4 Stable Models 10
3.5 Well-Founded Models 11
3.6 Partial Stable or Stationary Models 12
3.7 Relationships Between Different Semantics 12

4 Well-Founded and Stationary Models 13
4.1 Partial Models 13
4.2 Least Partial Models 17
4.3 The Quotient Operator 19
4.4 Well-Founded Models 21
4.5 Stationary Models 26
4.6 Well-Founded Model Coincides with the Smallest Stationary Model 27
4.7 Dynamic Stratification 28
4.8 Non-Herbrand Models 31
4.9 Relationship to Non-Monotonic Formalisms 33
4.10 Procedural Semantics: SLS-resolution 35

5 Conclusion 37

References 37

* Partially supported by the National Science Foundation grant #IRI-9313061. To appear in the Annals of Mathematics and Artificial Intelligence. Revised on June 6, 1994.

1 Introduction

The introduction and subsequent development of the formal foundations of logic programming and deductive databases 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. Consequently, a precise meaning or *semantics* must be associated with any logic or database program P in order to provide its declarative specification.

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*, 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 supplying 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 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 satisfies 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 content. 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.

Assuming that a logic program P is expressed in some language $\mathcal{L}$ and is considered to be a theory in some logic *Log* (e.g., classical predicate logic, three-valued logic or epistemic logic), the declarative semantics $SEM(P)$ of P is the set of all sentences in $\mathcal{L}$ which are considered to be true about P. It is natural to require that $SEM(P)$ be closed under logical consequence in *Log* and that it should at a minimum contain all sentences derivable from P in the given logic Log. However, in general, $SEM(P)$ contains many more sentences describing the *commonsense* consequences of P .

The semantics $SEM(P)$ 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 extension or completion $COMP(P)$, i.e., a (finite or infinite) theory in $\mathcal{L}$ extending P. For example, $COMP(P)$ can be Clark's predicate completion of P. A sentence S is then said to belong to $SEM(P)$ if and only if it derivable, in the logic Log, from the completion:

$$COMP(P) \models_{Log} S.$$

A closely related 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) in the logic Log. For example, $MOD(P)$ can be the set of all minimal models of P or the unique least model of P. A formula S is then said to belong to $SEM(P)$ if and only if it is satisfied in all intended models, i.e., if

$$MOD(P) \models_{Log} S \text{ (in particular, } M_P \models_{Log} S).$$

Although the proof-theoretic approach can be viewed as a *special case* of the model-theoretic approach it is often easier or more natural to use than the model-theoretic one.

2 Clark's Predicate Completion Semantics and its Drawbacks

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's 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[2]) set of all clauses

$$\begin{array}{l} q(T_1, \dots, T_n) \leftarrow V_1 \\ \qquad\qquad\dots \\ \qquad\qquad\dots \\ q(T_1, \dots, T_n) \leftarrow V_s \end{array}$$

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

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

Clark's predicate completion semantics $CLARK(P)$ of a program P is then defined[3] as the set of all sentences logically implied by Clark's completion $comp(P)$ of P, i.e., as the set of all sentences satisfied in all models of $comp(P)$.

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 legitimately use the following program P_1 to describe natural numbers:

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

When interpreted as a self-contained first order theory, the above program P_1 is too weak and does not reflect our intended meaning. In particular, it does not even imply that, say, *Mickey Mouse* (or anything else for that matter) is *not* a natural number. This is because what we really (intuitively) have in mind is the following theory:

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

which implies

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

and is in fact Clark's completion $comp(P_1)$ of the program P_1.

Unfortunately, Clark's semantics also has some serious drawbacks. More precisely, while Clark's predicate completion provides a suitable semantics for *recursion-free programs* [AB90], It often *fails to provide a proper semantics for programs involving recursion.*

Recursion in logic programming can occur in two essentially different forms, namely as:

---

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

[3] The obtained theory is also augmented with *Clark's Equality Axioms*, which include *unique names axioms* and *equality axioms* (see Section 4.8 or [PP90a] for more details). These axioms are only essential when considering the so called *non-Herbrand* models of Clark's completion and they will not play any role in the discussion that follows.

Positive recursion, which can be schematically illustrated by the program $p \leftarrow p$, in which the predicate p depends on a *positive* occurrence of itself;

Negative recursion, which can be schematically illustrated by the program $p \leftarrow \neg p$, in which p depends on a *negative* occurrence of itself.

Causes of the inadequate behavior of the Clark completion semantics for recursive programs are largely dependent on whether the recursion is negative or positive and therefore they are discussed separately below. We begin by discussing problems involving *negative* recursion, which are easier to correct.

2.1 The Negative Recursion Problem: Inconsistency

While Clark's completion $comp(P)$ is always consistent (and thus well-defined) for programs P *not* involving negative recursion (cf. [Cav89]), $comp(P)$ is frequently inconsistent for programs with negative recursion and therefore it *does not provide any reasonable semantics for such programs.*

For a trivial example, observe that Clark's completion of the program P_2

$$\begin{array}{l} a \\ b \leftarrow \neg a \\ p \leftarrow \neg p \end{array}$$

is

$$a; \quad b \leftrightarrow \neg a; \quad p \leftrightarrow \neg p,$$

which is inconsistent. As a result, the above program is not given any sensible meaning by the Clark semantics, in spite of the fact that it clearly should imply a and $\neg b$. This implies that the answer 'yes' to the query $\leftarrow a$, which would be returned by any Prolog system, is in fact *unsound*[4] with respect to Clark's semantics.

The next example is even more pathological. Observe that Clark's completion of the program P_3:

$$\begin{array}{l} a \\ p \leftarrow \neg q, \neg p \end{array}$$

is:

$$\begin{array}{l} a \\ p \quad \leftrightarrow \neg q \wedge \neg p \\ \neg q \end{array}$$

which is inconsistent. However, after adding to P_3 a clause $q \leftarrow q$ its completion becomes:

$$\begin{array}{l} a \\ p \leftrightarrow \neg q \wedge \neg p \\ q \leftrightarrow q \end{array}$$

[4] Unless we agree to assign Clark's semantics also to programs P for which $comp(P)$ is inconsistent, in which case the semantics will include *all* formulae and thus be totally useless.

which has a unique model in which a and q are true and p is false. However, the clause $q \leftarrow q$ represents a tautology which is clearly 'meaningless', i.e., it should not in any way affect the meaning of the program as it does not add any new information.

On the other hand, after adding to P_3 another 'meaningless' clause $p \leftarrow p$ its completion becomes:

$$\begin{array}{l} a \\ p \;\leftrightarrow p \vee (\neg q \wedge \neg p) \\ \neg q \end{array}$$

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

As a final example, let us consider the following program P_4 [Prz91c]:

$$\begin{array}{ll} work & \leftarrow \neg tired \\ sleep & \leftarrow \neg work \\ tired & \leftarrow \neg sleep \\ angry & \leftarrow \neg paid, work \\ paid & \leftarrow \end{array}$$

Clark's completion of the above program is:

$$\begin{array}{ll} work & \leftrightarrow \neg tired \\ sleep & \leftrightarrow \neg work \\ tired & \leftrightarrow \neg sleep \\ angry & \leftrightarrow \neg paid \wedge work \\ paid & \end{array}$$

which is easily seen to be inconsistent.

However, while it appears that the first three rules of the program P_4 describe only mutual relationships between propositions $tired, work$ and $sleep$, without providing sufficient information to determine their truth or falsity, it is clear that, regardless of the status of propositions $tired, work$ and $sleep$, the proposition $paid$ must be true and thus $angry$, by negation as failure, should be false. These are also the answers that would be returned by any Prolog system, but again, in spite of their intuitive naturality, they are essentially unsound with respect to Clark's semantics.

2.2 The Positive Recursion Problem: Insufficient Expressibility

As we mentioned before, Clark's semantics often fails to provide a suitable semantics for programs involving positive recursion. This problem applies both to standard Clark's semantics $CLARK(P)$ as well as to its Fitting-Kunen extension $PCLARK(P)$ discussed later in this chapter. The problem has been extensively discussed in the literature (see e.g. [She88, She84, Prz89b, VGRS90]). We illustrate it on the following three examples.

Example 1. Suppose that to the program P_1 defined before we add a seemingly meaningless clause:

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

Intuitively, 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 $Mickey\ Mouse$ (or anything else, for that matter) is *not* a natural number. □

Example 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 positive program P_5:

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

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 cannot be derived. Here, the difficulty is caused by the existence of symmetric clauses $edge(c,d)$ and $edge(d,c)$. Removal of at least one of these edges eliminates the problem. □

Example 3. Suppose that program P_6 is given by the following clauses:

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

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_6 yields

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

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 basic problem is that the addition of positive recursion to a program often leads to a completion, which is *too weak to express the intended meaning of the program*, i.e., a completion from which some intuitively obvious conclusions can no longer be derived. In particular, this problem is responsible for the fact that Clark's semantics *is not sufficiently expressive* to naturally represent transitive closures [Kun88].

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 contradicts the very spirit of logic programming and thus should be rejected.

3 Eliminating Drawbacks of Clark's Semantics

In this section we briefly discuss several recently proposed semantics of logic programs which attempt to eliminate the drawbacks of Clark's semantics discussed above. For a more thorough discussion the reader is referred to [PP90a]. Next section will be devoted to a detailed study of the well-founded and stationary semantics.

3.1 Partial Clark's Predicate Completion Semantics

As shown by [Fit85] and [Kun87], the inconsistency problem for Clark's semantics, which is caused by *negative* recursion, can be elegantly eliminated by replacing Clark's semantics by its extension $PCLARK(P)$ obtained by considering all *partial* models of Clark's completion $comp(P)$ instead of using only *total* models.

More precisely, the idea is to consider interpretations and models in which the truth values *true* and *false* are not necessarily assigned to *all* (ground) atoms but only to *some* of them, leaving the truth value of the remaining atoms *undefined.* The assignment of truth values given by a partial interpretation I is then naturally extended to the class of all sentences (closed formulae). For example, the truth value of a disjunction $F \vee G$ is defined as *true*, if one of F or G is true in I, *false*, if both F and G are false in I and is *undefined* otherwise. Similarly, the truth value of $\neg F$ is *true* if F is false in I, is *false* if F is true in I and is *undefined* otherwise. The equivalence $F \leftrightarrow G$ is defined to be true if both F and G have the same truth value and is defined to be false otherwise.

As we have seen above in the case of programs P_2, P_3 and P_4, a logic program may contain predicates whose truth or falsity is *not fully determined* by the program (and thus is *undefined*), in addition to predicates whose truth values are *completely determined* by the program. Partial models enable us to assign sensible semantics to such programs, without loosing potentially valuable information contained in them. The need to consider partial models (possible worlds) to describe our knowledge naturally follows from the fact that our knowledge about the world is almost always incomplete and therefore we need the ability to describe possible worlds (models) in which some facts are neither true nor false and thus their status is undefined.

The *partial Clark predicate completion* semantics $PCLARK(P)$ of a program P is defined as the set of all sentences satisfied in all *partial* models of Clark's completion comp(P) of P. In other words, we consider all logical consequences of Clark's completion in a suitable *3-valued logic*. The new semantics no longer suffers from the inconsistency problem, caused by negative recursion, because, as shown in [Fit85], Clark's completion $comp(P)$ of *any* logic program P always has at least one partial model. The partial Clark completion semantics is therefore well-defined for any logic program P. Since any sentence from $PCLARK(P)$ must hold in all partial models of $comp(P)$, and thus also in all total models of $comp(P)$, the partial Clark semantics is *weaker* than the original Clark semantics, i.e. for any logic program P we have:

$$PCLARK(P) \subseteq CLARK(P).$$

For example, Clark's completion $comp(P_2)$ of the program P_2 has a unique partial model[5] M, in which the proposition a is true and b is false but the status of p is undefined. Consequently, the partial Clark's semantics $PCLARK(P_2)$ of P_2 implies that a is true and b is false but leaves the status of p undefined.

Similarly, Clark's completion of the program P_3 has exactly one partial model, in which a is true and q is false but p is not defined. Therefore, the semantics $PCLARK(P_3)$ of P_3 implies that a is true, q is false and leaves the status of p undefined.

Finally, Clark's completion of the program P_4 also has a unique partial model in which *paid* is true, *angry* is false and *sleep*, *tired* and *work* are not defined and therefore, as one would expect, $PCLARK(P_4)$ implies *paid* and $\neg angry$, but it does not determine the status of *work*, *tired* and *sleep*.

As we can see, in those three cases, the meaning assigned by the partial Clark semantics agrees with our intuitions. The partial Clark completion semantics, proposed by Fitting and Kunen, not only eliminates the inconsistency problem but also displays a much more regular behavior than the original semantics and therefore it may be viewed as the *"true" Clark semantics.*

Unfortunately, as the reader can easily verify, the partial Clark completion semantics does *not* avoid the expressibility problems involving *positive* recursion that were discussed in the previous section.

3.2 Least Models

The classical result proved in [VEK76] shows that every *positive logic program* P, i.e., a program in which all premises are positive, has a unique least (Herbrand)

[5] Here we restrict our attention to Herbrand models only.

model M_P. This important result immediately leads to the definition of the so called *least model semantics* $LEAST(P)$ for positive programs. A sentence F belongs to $LEAST(P)$ if and only if F is true in the least model M_P of P.

The least model semantics is very intuitive, eliminates the positive recursion problems suffered by Clark's semantics, and, since it only applies to positive programs, it obviously is not affected by the negative recursion problem. The semantics properly reflects the intended meaning of positive logic programs and is motivated by the idea that we should minimize positive information as much as possible, limiting it to facts explicitly 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* [Rei78].

Since the least model of a positive program is also a model of Clark's completion, the least model semantics is *stronger* than Clark's semantics. More precisely, for any positive program P we have:

$$CLARK(P) = PCLARK(P) \subseteq LEAST(P).$$

Unfortunately, the least model semantics is well defined only for the class of positive programs. Programs which are not positive, in general, do not have least models and therefore the least model semantics is not applicable to such programs.

3.3 Perfect Models

In [ABW88] and [VG89b] the important class of *stratified programs* was introduced and a natural *semantics* for this class of programs was proposed. The class of stratified programs includes all positive programs and – imprecisely speaking – consists of those programs in which no predicate symbol depends negatively on itself. In particular, stratified programs *do not allow negative recursion.* The proposed semantics has been further investigated in [Prz88a, Prz89b], where it was characterized by means of preferred models, given the name of the *perfect model semantics* and extended to a broader class of *locally stratified* programs. We denote the perfect model semantics of a program P by $PERF(P)$.

The introduction of stratified programs and the perfect model semantics constituted a major breakthrough, by providing a large class of programs, which allow negation and positive recursion and yet admit a natural semantics which extends the least model semantics of positive programs and eliminates the positive recursion problem present in Clark's semantics Naturally, the negative recursion problem does not apply to stratified programs. It appears that the perfect model semantics has since been almost universally accepted as a *correct semantics for the class of (locally) stratified programs.*

The perfect model semantics is based on the idea of constructing an *iterated* least model of a program or – in other words – applying the iterated closed world assumption [GPP89]. It can also be equivalently defined by using the iterated fixed point approach or the preference relation between predicate symbols. As it was the case with the least model, the perfect model of a (locally) stratified program is also a model of Clark's completion and thus the perfect model semantics is stronger than Clark's semantics. More precisely, for any (locally) stratified program P we have:

$$CLARK(P) = PCLARK(P) \subseteq PERF(P).$$

Moreover, for any positive logic program $LEAST(P) = PERF(P)$.

The perfect model semantics is a *special case* of more general *stable*, *stationary* and *well-founded* semantics which are discussed next.

The only drawback of the perfect model semantics is the fact that it is restricted to the class of (locally) stratified programs. Several researchers pointed out that there exist interesting and useful logic programs with natural intended semantics, which are not locally stratified [GL88, VGRS90].

3.4 Stable Models

The *stable model semantics* was introduced in [GL88] in a successful attempt to extend the perfect model semantics to a class of programs significantly broader than the class of (locally) stratified programs. The stable model semantics $STABLE(P)$ has an elegant and simple fixed point definition and is closely related to the autoepistemic logic. An equivalent definition of stable models based on default logic was independently introduced in [BF91]. Unfortunately, while eliminating the positive recursion problem, the stable model semantics *suffers from essentially the same negative recursion problem as Clark's predicate completion semantics.* Namely, first of all, it is not defined for many programs involving negative recursion, thus making it impossible to extract from them all the potentially valuable information. For example, as it was the case with Clark's semantics, the stable model semantics is not defined for programs P_2, P_3 and P_4 described in Section 2.1. Moreover, even in those cases when the stable semantics is defined it does not always lead to the expected (intended) meaning of the program [VGRS90, PP90a], as shown by the following example.

Example 4. [VGRS90] Let P_7 be given by:

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

This program has a unique stable model $M = \{p, b\}$, which is also the unique (total) model of Clark's completion $comp(P_7)$, and therefore both the stable model semantics $STABLE(P_7)$ and Clark's predicate completion semantics $CLARK(P_7)$ imply that p and b are true and a is false. This conclusion is viewed by many researchers as unintuitive. Indeed, although the first two clauses do not give any preference to either b or a, the proposition b is made true because: (1) p must be true because (in 2-valued logic) this is the only way of satisfying the third clause whose premise is $\neg p$; (2) the only way to justify p being true is to assume that $\neg a$, and thus also b, is true. In other words, the reasoning essentially proceeds *backwards*, namely, since p must be true we must assume that a is false. This appears to violate the usual, *forward* way of reasoning in logic programming. As a result, it is easy to see that it is impossible to derive b from P_7 using any form of *Horn-resolution*. This is because any Horn-resolution beginning with the goal b will reach only the first two clauses of P, from which b cannot be derived.

On the other hand, partial Clark's semantics $PCLARK(P_7)$ of P_7 has three models. One of them is the stable model M mentioned above, in the second a is true, b is false and p is undefined and in the third all a, b and p are undefined. As a result, the partial Clark semantics remains undecided as far as answers to queries about a, b and p are concerned, which seems to properly reflect the incomplete (or uncertain) information contained in the program. □

Stable Semantics is not Cumulative A semantics $SEM(P)$ is called *cumulative* [Mak88] if, for any program P and atoms A and B, from the fact that $SEM(P) \models_{Log} A$ and $SEM(P) \models_{Log} B$ it follows that $SEM(P \cup A) \models_{Log} B$. The above example illustrates another important drawback of the stable model semantics, namely, the fact that it is *not* cumulative. Indeed, even though the stable model semantics $STABLE(P_7)$ of P_7 implies p and b, the stable model semantics of the program $P_7^* = P_7 \cup \{p\}$, obtained by adding p to P_7, *no longer* implies b. The lack of cumulativity is likely to cause serious maintenance problems. Indeed, as a result of adding to a program P a fact A derivable from P, we may be forced to retract another derivable fact B.

The stable model semantics is also computationally intractable. As it was proven in [MT88], just determining whether a finite datalog program with negation[6] *has* a stable model is *NP-complete*, whereas the problem of computing the intersection of all stable models of such simple programs is *co-NP-hard* [KS89].

While the stable semantics exhibits good behavior under *positive recursion*, its behavior under *negative* recursion is essentially *identical* to the behavior of the (standard) Clark completion semantics $CLARK(P)$. In this sense, the stable semantics $STABLE$ is *dual* to the partial Clark semantics $PCLARK(P)$, which ensures a proper treatment of negative recursion but fails to ensure a proper treatment of positive recursion.

3.5 Well-Founded Models

The *well-founded semantics*, $WF(P)$, introduced in [VGRS90], is also an extension of the perfect model semantics from the class of (locally) stratified logic programs to the class of normal logic programs. However, in contrast to the stable semantics, the well-founded semantics is defined for *all* normal logic programs. It combines the features of both the *stable* and *partial Clark* semantics

[6] I.e.,a program which does not use functional symbols, or, more precisely, one whose ground instantiation is finite.

and thus ensures a proper behavior under both positive and negative recursion. As a result, the well-founded semantics provides a natural and intuitive semantics for the class of all logic programs. It also shares many of the natural properties of the perfect model semantics [Prz89a]. Namely, well-founded models can be equivalently defined as *iterated least models* and as *iterated fixed points.* The well-founded model also leads to a natural notion of *dynamic stratification* of an arbitrary logic program.

Well-founded semantics has been shown to have a sound and complete proof procedure, called *SLS-resolution* [Prz89a, Ros89], and several natural constructive characterizations [Prz89a, VG89a, Bry89, Prz94b]. Recently, D. S. Warren introduced the Extended Warren Abstract Machine (XWAM) for this semantics [War89] and developed an elegant interpreter in Prolog [CW92]. For datalog programs with negation, the computation of well-founded models is *quadratic* in the size of the program [VGRS90]. Moreover, the well-founded semantics is always cumulative [Dix91, LY91].

3.6 Partial Stable or Stationary Models

Stable and well-founded semantics are closely related. In [Prz90, Prz91c] the author introduced *partial stable models* of normal logic programs, which were later renamed *stationary models* [Prz91b]. Partial stable (or stationary) models constitute a natural generalization of the (total) stable models and are defined as fixed points of a program transformation (factorization) which is analogous to the transformation used in the original definition of stable models [GL88]. It turns out that the well-founded model always coincides with the *smallest partial stable model* and thus every normal logic program P has at least one partial stable model. It also follows that the well-founded semantics of an arbitrary normal logic program always coincides with the *partial stable semantics*, $PSTABLE(P)$, or – equivalently – with the *stationary semantics*, $STAT(P)$, defined as the set of all sentences true in all partial stable (or all stationary) models of a program P:

$$WF(P) = PSTABLE(P) = STAT(P).$$

One can thus say that the only difference between the stable and well-founded semantics is the fact that the former uses only *total* stable models while the latter allows all *partial* stable models. Consequently, the relationship between the stable semantics $STABLE(P)$ and the partial stable or well-founded semantics $PSTABLE(P) = WF(P)$ is analogous to the relationship between Clark's completion semantics $CLARK(P)$ and partial Clark completion semantics $PCLARK(P)$.

Partial stable, and, in particular, well-founded models are also closely related to non-monotonic formalisms in AI. It has been proven in [Prz91d] that partial stable models of an arbitrary logic program are in a one-to-one correspondence to natural 3-valued forms of all four major formalizations of non-monotonic reasoning in AI: McCarthy's circumscription, Reiter's closed world assumption, Moore's autoepistemic logic and Reiter's default theory. Recently, the author showed [Prz94a, Prz91a] that partial stable models can be equivalently defined (without any reference to partial models) as expansions in autoepistemic logic and that essentially *the only difference* between stable and partial stable models is the fact that the first are based on Reiter's Closed World Assumption CWA while the latter are based on Minker's $GCWA$ (or, equivalently, on McCarthy's Circumscription $CIRC$). Similar results were obtained for default logic [PP94]. The relationship between partial stable models and non-monotonic formalisms is discussed in more detail in Section 4.9.

Partial stable, and thus also well-founded, models have been recently generalized to the class of all *disjunctive* logic programs [Prz94b, Prz91b].

3.7 Relationships Between Different Semantics

The following table summarizes the basic features of various semantics discussed in this section. It specifies whether a given semantics properly handles positive and negative recursion and describes the class of programs for which the semantics is defined.

Semantics	Pos. Recursion	Neg. Recursion	Class of Progs.
Clark	No	No	Restricted
Partial Clark	No	Yes	All
Least	Yes	Yes	Positive
Perfect	Yes	Yes	(Loc.) Stratified
Stable	Yes	No	Restricted
Well-Founded (= Partial Stable = Stationary)	Yes	Yes	All

The following inequalities describe the remaining relationships between the semantics listed above. All of them apply only to those classes of programs for which they are *defined*. The notation $SEM_1|Class = SEM_2$ means that semantics SEM_1 coincides with semantics SEM_2 in the class $Class$ of programs .

$$\begin{array}{ccc} WF(P) = STAT(P) = PSTABLE(P) & \subseteq & STABLE(P) \\ \cup\!\!| & & \cup\!\!| \\ PCLARK(P) & \subseteq & CLARK(P). \end{array}$$

$$WF(P)|\text{locally stratified} = STABLE(P)|\text{locally stratified} = PERF(P).$$

$$PERF(P)|\text{positive} = LEAST(P).$$

The above table shows that partial Clark's completion semantics is the weakest of all semantics considered in here. The table does not contain any relationship between the Clark completion semantics $CLARK(P)$ and the well-founded semantics $WF(P) = PSTABLE(P)$; the two semantics are in fact incompatible.

It follows from the above tables that the partial Clark predicate completion semantics $PCLARK(P)$ can be viewed as a natural (and computationally less expensive) *approximation* to the well-founded (or partial stable) semantics $WF(P) = PSTABLE(P)$ of logic programs, in the sense that any answers given by $PCLARK(P)$ (in particular, those computed by the SLDNF-resolution) are correct with respect to the well-founded semantics (but not vice versa!). Similarly, the Clark predicate completion semantics $CLARK(P)$ can be viewed as a natural *approximation* to the stable semantics $STABLE(P)$.

4 Well-Founded and Stationary Models

4.1 Partial Models

In this section we define *logic programs* and their *partial interpretations and models.* We closely follow the approach developed in [Prz89a, PP90a].

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. In addition, any alphabet is assumed to contain a countably infinite set of *variable* symbols, the connectives $\wedge, \vee, \neg, \leftarrow$ and quantifiers $\exists, \forall$ and the usual punctuation symbols. Moreover, we assume that our alphabet also contains propositional symbols $\mathbf{t}$, $\mathbf{u}$ and $\mathbf{f}$, denoting the properties of being *true* (respectively, *undefined* or *false*). 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 either an atom A or its negation $\neg A$. An expression is called *ground* if it does not contain any variables. If G is a quantifier-free formula, then by a *ground instance* of G we mean any ground formula obtained from G by uniformly substituting ground terms for all variables. For a given formula G of $\mathcal{L}$ its *universal closure* or just *closure* is obtained by universally quantifying all variables in G which are not bound by any quantifier.

Definition 1. By a *partial Herbrand interpretation* I of the language $\mathcal{L}$ we mean any *set of ground literals.* We assume that every interpretation I contains $\mathbf{t}$ and $\neg\mathbf{f}$, but does *not* contain either $\mathbf{u}$ or $\neg\mathbf{u}$. An interpretation is *consistent* if there is no atom A such that both A and $\neg A$ belong to I. Unless stated otherwise all interpretations will be assumed to be consistent. An interpretation is *total* if for every ground atom A (except $\mathbf{u}$) either A or $\neg A$ belongs to I. □

An interpretation I will be usually written in the form $I = Pos(I) \cup Neg(I)$, where $Pos(I)$ contains all positive literals (atoms) in I and $Neg(I)$ contains the set of all negative literals in I. In other words, $Pos(I)$ represents the set of atoms which are *true* in I (i.e., those that belong to I) and $Neg(I)$ represents the set of atoms which are *false* in I (i.e., those whose negation belongs to I). Throughout the paper, we consider only *Herbrand* interpretations and models, although our results can be easily extended to non-Herbrand models.

Suppose that I is an interpretation and F and G are arbitrary closed formulae (i.e., *sentences*). We now recursively define what it means for a sentence to be *true* or *false* in I:

- A ground atom A is true in I if it belongs to I and it is false in I if $\neg A$ belongs to I.
- $\neg F$ is true (respectively, false) in I if and only if F if false (respectively, true) in I.
- $F \vee G$ is true (respectively, false) in I if and only if either F or G is true in I (respectively, if both F and G are false in I).
- $F \wedge G$ is true (respectively, false) in I if and only if both F and G are true in I (respectively, if either F or G is false in I).
- $F \leftarrow G$ is true in I if the truth value of F is greater than or equal to the truth value of G (with the truth values ordered by $\mathbf{f} < \mathbf{u} < \mathbf{t}$). Otherwise, $F \leftarrow G$ is false in I.
  In other words, $F \leftarrow G$ is true if F is true in I whenever G is true in I and if G is false in I whenever F is false in I. Otherwise, $F \leftarrow G$ is false in I.
- $\exists x\ F(x)$ is true in I if one of ground instances of $F(x)$ is true in I and it is false in I if all of its ground instances are false in I (as usual, we assume that x is the only free variable in $F(x)$).
- $\forall x\ F(x)$ is true in I if all ground instances of $F(x)$ are true in I and it is false in I if one of its ground instances is false in I.

The connective $\leftarrow$ represents what we call a *constructive implication.* It is easy to see that the formula $F \leftarrow G$ is *not* equivalent to $F \vee \neg G$. Indeed, $F \leftarrow G$ is always true if G is *not* true and F is not false, while $F \vee \neg G$ is true if and only if either F is true or G is false. The constructive implication connective $\leftarrow$ will be used in the definition of logic programs.

Definition 2. If a closed formula F is true in a (partial or total) interpretation M then we write $M \models F$ and say that M is a *model* of F or that F is *satisfied* in M. By a *theory* T we mean a (finite or infinite) set of closed formulae. A (partial or total) interpretation M is a (partial or total) *model* of a theory T if all formulae from T are satisfied in M, i.e., if $M \models F$, for all F in T.

We say that a theory T *logically implies* a sentence F in classical (respectively, 3-valued) logic if F is satisfied in all total (respectively, partial) models M of T. We denote this fact by $T \models F$ (respectively, $T \models_p F$). □

Example 5. Suppose that $I = \{A, \neg C\}$ is an interpretation[7]. Then:

- $A \vee B$, $\neg C \vee \neg B$ and $A \wedge \neg C$ are true, $\neg A \vee C$ and $\neg A \wedge B$ are false, but $B \vee \neg B$ and $A \wedge B$ are neither true nor false.
- $B \leftarrow \neg B$, $B \leftarrow \mathbf{u}$, $A \leftarrow \mathbf{u}$, $B \leftarrow B$ and $A \leftarrow B$ are true, but $\neg A \leftarrow B$, $C \leftarrow \mathbf{u}$ and $C \leftarrow A$ are false. □

[7] We ignore literals $\mathbf{t}$ and $\neg\mathbf{f}$.

Definition 3. By a *logic program* we mean a set of universally closed *clauses* of the form

$$A \leftarrow B_1 \wedge \ldots \wedge B_m \wedge \neg C_1 \wedge \ldots \wedge \neg C_n,$$

where $m, n \geq 0$ and A, B_i's and C_j's are atoms. □

We allow B_i's and C_j's to be either one of t , u or f . Observe, however, that only the presence of the *undefined* proposition u is essential, because any clause containing a false premise f can simply be removed and all true premises t can always be erased.

Conforming to a standard convention, conjunctions are replaced by commas and therefore clauses are written in the form

$$A \leftarrow L_1, \ldots, L_k,$$

where L_i's are literals. A program is called *positive* if none of its clauses contains negative premises or unknown premises u . Programs not allowing negative premises but allowing undefined premises u will be called *non-negative.*

If P is a program then, unless stated otherwise, we assume that the alphabet $\mathcal{A}$ used to write P consists precisely of all the constant, predicate and function symbols that explicitly *appear* in P and thus the language $\mathcal{A} = \mathcal{A}_P$ is completely determined [8] by the program P. The following proposition is immediate.

Proposition 4. *A (partial or total) interpretation M is a model of a program P if and only if for all ground instances of its clauses*

$$A \leftarrow L_1, \ldots, L_k,$$

if all L_i's are true in M then A is true in M and if A is false in M then at least one of the L_i's is also false in M. □

By a *ground instantiation* of a logic program P we mean the (possibly infinite) theory consisting of all ground instances of clauses from P. It is easy to see that a Herbrand interpretation M is a model of a program P if and only if it is a model of its ground instantiation. Therefore, 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.

There are *two natural orderings* between interpretations, one of them, $\preceq$, is called the *truth ordering* and the other, $\subseteq$, is called the *information ordering.* The latter coincides with the set-theoretic *inclusion.*

Definition 5. [Prz89a] Suppose that $I = Pos(I) \cup Neg(I)$ and $J = Pos(J) \cup Neg(J)$ are two interpretations. We define:

$$I \preceq J \text{ if } Pos(I) \subseteq Pos(J) \text{ and } Neg(I) \supseteq Neg(J);$$

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

and

$$I \subseteq J \ \text{ if } \ Pos(I) \subseteq Pos(J) \ \text{ and } \ Neg(I) \subseteq Neg(J).$$

Models which are *least* in the sense of the truth ordering $\preceq$ will be simply called *least models*. On the other hand, models which are *least* in the sense of inclusion $\subseteq$ will be called *smallest* models. □

The two orderings are significantly different. While least models and interpretations M of a program P minimize the *degree of truth* of atoms, by minimizing the set $Pos(M)$ of true atoms and maximizing the set $Neg(M)$ of false atoms, models and interpretations which are smallest in the sense of inclusion minimize the *degree of information* of their atoms, by jointly minimizing the sets $Pos(M)$ and $Neg(M)$ of atoms which are defined as either true or false and thus maximizing the set of atoms, whose truth value is undefined.

For example, the smallest model of the program $p \leftarrow p$ is obtained when p is undefined, while its least model is obtained when p is false. Similarly, the *smallest partial interpretation* is the empty interpretation $J_0 = \{\}$ in which all atoms (except t and f) are undefined, while the *least partial interpretation* is the interpretation I_0 in which all atoms (except t and u) are false.

4.2 Least Partial Models

In this section we introduce the following generalization of the classical Van Emden - Kowalski result [VEK76], which states that every non-negative logic program has a unique least partial model. We use it in Section 4.4 where we characterize well-founded models as iterated least partial models of logic programs.

Theorem 6. *[Prz91c] Every* non-negative *logic program P has a unique least partial model $LPM(P)$.* □

The classical Van Emden - Kowalski result [VEK76] states that every *positive* logic program has a unique least *total* model. Due to the fact that non-negative programs allow the undefined propositional symbol u to occur among premises of program clauses, they are strictly *more general* than positive programs. Consequently, least models of non-negative programs may not be *total* but rather *partial* models.

Example 6. Suppose that the non-negative program P is given by:

$$\begin{aligned} &c \leftarrow \\ &a \leftarrow c, u \\ &b \leftarrow b, u \end{aligned}$$

The least partial model of P is $M = \{c, \neg b\}$, i.e., in M the atom c is true, b is false and a is undefined. The least partial model M minimizes the truth values of its atoms as much as possible while still satisfying all program clauses. Observe, that a cannot be made false because the truth value of the conjunction of premises in the second clause is u and therefore the truth value of a has to be at least u in order for the second clause to be satisfied in M. □

As it was the case with the Van Emden - Kowalski least total model, the least partial model $LPM(P)$ can be obtained as the least fixed point of the following natural *immediate consequence operator* $\widehat{T}$, which acts on the set of all partial interpretations of a program and generalizes the Van Emden-Kowalski immediate consequence operator T [VEK76]. The definition below assumes that the program has already been instantiated.

Definition 7. [Prz91c] Suppose that P is a logic program and I is a partial interpretation of P. Define $\widehat{T}(I)$ to be the partial interpretation which contains a literal:

(i) A if and only if there is a clause $A \leftarrow A_1, \ldots, A_n$ in P, all of whose premises A_i are true in I;
(ii) $\neg A$ if and only if for every clause $A \leftarrow A_1, \ldots, A_n$ in P, there is a premise A_i which is false in I, i.e., such that $\neg A_i$ belongs to I. □

The following theorem gives a *fixed-point characterization* of least partial models of non-negative programs. Recall that we denote by I_0 the *least partial interpretation*, i.e., the interpretation in which all atoms (except t and u) are false.

Theorem 8. *[Prz91c] If P is a non-negative program, then the operator $\widehat{T}$ has the least fixed point which coincides with the least partial model $LPM(P)$ of P, i.e., $LPM(P)$ is the least interpretation I such that $\widehat{T}(I) = I$.*

Moreover, $LPM(P)$ can be obtained by iterating the operator $\widehat{T}$ (at most) ω times. More precisely, the sequence $I_n = \widehat{T}^{\uparrow n}(I_0)$, $n = 0, 1, 2, \ldots, \omega$, of interpretations obtained by iterating $\widehat{T}$ beginning with the interpretation I_0 in which all atoms are false, is monotonically increasing, with respect to the truth ordering $\preceq$, and has a fixed point $I_\omega = \bigcup_{n<\omega} I_n = \widehat{T}^{\uparrow \omega}(I_0)$, which coincides with the least partial model $LPM(P)$.

In addition, if P is a datalog *program, i.e., if it does not contain non-ground functional terms, or - equivalently - if the instantiated program is finite, then the above described construction of the least partial model of P stops after* finitely *many steps and is* linear *in the size of the program.* □

Example 7. For the above program P:

$$\begin{aligned} &c \leftarrow \\ &a \leftarrow c, u \\ &b \leftarrow b, u \end{aligned}$$

we obtain:

$$I_0 = \{\neg a, \neg b, \neg c\};$$
$$I_1 = \widehat{T}(I_0) = \{\neg a, \neg b, c\};$$
$$I_2 = \widehat{T}(I_1) = \{\neg b, c\};$$
$$I_3 = \widehat{T}(I_2) = I_2 = \{\neg b, c\};$$

and therefore the least partial model $LPM(P)$ of P coincides with the least fixed point $I_2 = \{\neg b, c\}$ of $\widehat{T}$. □

The following easy proposition describes the simple relationship existing between the least partial model $LPM(P)$ of a non-negative program P and the least (total) models $LM(P_{u \leftarrow f})$ and $LM(P_{u \leftarrow t})$ of positive programs $P_{u \leftarrow f}$ (respectively, $P_{u \leftarrow t}$) obtained from P by replacing all undefined premises u in P by false (respectively, true) premises f (respectively, t). It also shows how one can compute least partial models of P by computing least models of the two derived *positive* programs.

Proposition 9. *For any non-negative logic program the following equalities hold:*

$$Pos(LPM(P)) = Pos(LM(P_{u \leftarrow f}))$$

$$Neg(LPM(P)) = Neg(LM(P_{u \leftarrow t})).\Box$$

Consequently, the positive atoms of the least partial model $LPM(P)$ of P are precisely the positive atoms in the *least model* $LM(P_{u \leftarrow f})$ of the program $P_{u \leftarrow f}$ and the negative atoms of the least partial model $LPM(P)$ of P are precisely the negative atoms[9] in the least model $LM(P_{u \leftarrow t})$ of the program $P_{u \leftarrow t}$

4.3 The Quotient Operator

Before giving a constructive characterization of well-founded models as iterated least partial models of logic programs we first have to recall the *quotient operator* $\frac{P}{I}$ defined in [Prz91c] which assigns to any logic program P and to any partial interpretation I the unique non-negative program $P' = \frac{P}{I}$. This operator extends to partial interpretations the Gelfond-Lifschitz transformation defined in [GL88].

Definition 10. [Prz91c] Let P be a logic program and let I be any partial interpretation. By the *quotient of P modulo I* we mean the new program $\frac{P}{I}$ obtained from P by replacing in every clause of P all *negative* premises $\neg C$ which are true (respectively, undefined; respectively, false) in I by their corresponding truth values t (respectively, u or f). $\Box$

As we pointed out in Section 4.1, if t appears among the premises of a given clause then it can be simply erased (ignored) and if f appears among the premises of a given clause then the *whole clause* can be erased without changing anything. On the other hand, the u 's in general cannot be removed. This immediately leads to the following important corollary:

Corollary 11. *[Prz91c] For any logic program P and any partial interpretation I the quotient $\frac{P}{I}$ of P modulo I is* always *a non-negative program and therefore, by Theorem 6, it has a unique least partial model:*

$$LPM(\frac{P}{I}).\Box$$

[9] I.e., the complement of the set of positive atoms in the least model.

For a fixed program P, we will denote by Ψ the *operator* assigning to any partial interpretation I the least partial model $LPM(\frac{P}{I})$ of the quotient program $\frac{P}{I}$:

$$\Psi(I) = LPM(\frac{P}{I}).$$

Example 8. Consider the program P_2 discussed in Section 2.1:

$$\begin{array}{l} a \\ b \leftarrow \neg a \\ p \leftarrow \neg p \end{array}$$

and let I be the partial interpretation $\{a,\ \neg b\}$, i.e., the interpretation in which a is true, b is false and p is undefined. Then the quotient $\frac{P_2}{I}$ of P_2 modulo I is given by:

$$\begin{array}{l} a \\ b \leftarrow f \\ p \leftarrow u \end{array}$$

and since the second clause contains a false premise it can be removed resulting in the non-negative program:

$$\begin{array}{l} a \\ p \leftarrow u\ . \end{array}$$

Observe, that the least partial model $LPM(\frac{P_2}{I})$ of $\frac{P_2}{I}$ is I itself. Consequently, $\Psi(I) = I$ and I is the *fixed point* of the operator Ψ.

Recall that we denote by J_0 the empty interpretation $\{\}$, in which all propositions (except t and f) are undefined, i.e., the *smallest partial interpretation* in the sense of inclusion. The quotient $\frac{P_2}{J_0}$ of P_2 modulo J_0 is given by:

$$\begin{array}{l} a \\ b \leftarrow u \\ p \leftarrow u\ . \end{array}$$

and its least partial model $LPM(\frac{P_2}{J_0})$ is the model $\{a\}$, in which a is true and b and p are undefined. □

The intuition behind the notion of a quotient program $\frac{P}{I}$ and the notion of its least partial model $LPM(\frac{P}{I})$ is very simple and natural. Imagine that the partial interpretation I represents our current, perhaps not yet complete, *knowledge* about the program, encoded by specifying propositions about which we already know that they are true or false in P (initially, we have no such knowledge at all and therefore the initial interpretation I to start with is the empty interpretation J_0 in which all propositions are undefined).

The construction of the quotient program $\frac{P}{I}$ of P modulo I allows us to incorporate our current knowledge (encoded in I) into the program P, by means of replacing all negative premises $\neg C$ in P by their corresponding truth values in I. The computation of the least partial model $LPM(\frac{P}{I})$ of the new program

$\frac{P}{I}$ obtained in this way, allows us to possibly augment our knowledge with new information, obtained by finding out what additional propositions are now *forced to be true* by the information contained in I and which can be *assumed false* in view of the information contained in I. The process of determining the propositions which can be assumed false, the least partial model represents the *closed world assumption*, which is indispensable to the proper interpretation of negation by default $\neg C$ in logic programs.

Example 9. Suppose that the program P_8 is given by:

$$\begin{array}{l} b \leftarrow \neg a \\ c \leftarrow \neg b, p \\ p \leftarrow \neg p \end{array}$$

Initially, we have no knowledge available about the program. Therefore our initial knowledge is best represented by the empty interpretation $J_0 = \{\}$, in which all propositions are undefined. The quotient $\frac{P_8}{J_0}$ of P_8 modulo J_0 is given by:

$$\begin{array}{l} b \leftarrow u \\ c \leftarrow u\ , p \\ p \leftarrow u \end{array}$$

and its least partial model $J_1 = \Psi(J_0) = LPM(\frac{P_8}{J_0})$ is the model $\{\neg a\}$, in which a is false and all the other propositions are undefined. The interpretation J_1 adds a new piece of information about P_8, namely the fact that a can be safely assumed false, by virtue of the closed world assumption.

We repeat the procedure starting with the interpretation J_1 which now represents our current knowledge. The quotient $\frac{P_8}{J_1}$ of P_8 modulo J_1 is given by:

$$\begin{array}{l} b \leftarrow t \\ c \leftarrow u\ , p \\ p \leftarrow u \end{array}$$

and its least partial model $J_2 = \Psi(J_1) = LPM(\frac{P_8}{J_1})$ is the model $\{\neg a,\ b\}$, in which a is false, b is true and the propositions c and p are undefined. The interpretation J_2 adds one more piece of information about P_8, namely the fact that b now has to be true because a is false.

We repeat the procedure again starting with the interpretation J_2 which now represents our current knowledge. The quotient $\frac{P_8}{J_2}$ of P_8 modulo J_2 is given by:

$$\begin{array}{l} b \leftarrow t \\ c \leftarrow f\ , p \\ p \leftarrow u\ . \end{array}$$

Since the second clause can be removed, the least partial model $J_3 = \Psi(J_2) = LPM(\frac{P_8}{J_2})$ of the resulting non-negative program is the model $\{\neg a,\ b,\ \neg c\}$, in which a and c are false, b is true and p is undefined. The interpretation J_3 further increments our knowledge about P_8, by establishing the fact that also c can be safely assumed false, by virtue of the closed world assumption.

If we repeat the above procedure starting with the interpretation J_3 which represents our current knowledge, we discover that the quotient $\frac{P_8}{J_3}$ of P_8 modulo J_3 is:

$$\begin{aligned} &b \leftarrow t \\ &c \leftarrow f\,, p \\ &p \leftarrow u \end{aligned}$$

and is exactly identical to the one obtained in the previous step. Consequently, also the least partial model $LPM(\frac{P_8}{J_3})$ of $\frac{P_8}{J_3}$ is the same, namely J_3 itself. We have therefore reached a fixed point, i.e., $J_3 = \Psi(J_3)$, and we can no longer add any new information to our knowledge about the program P_8 and therefore we can reasonably view the partial model $J_3 = \{\neg a,\ b,\ \neg c\}$ as representing *as complete as possible knowledge* about the program. □

Observe that the above construction closely resembles the method used to define *perfect models* of *stratified programs* (see [ABW88, VG89b, Prz88a]), but, as we will show in the next section, it works for *all logic programs* and computes their *well-founded models.*

4.4 Well-Founded Models

In this section we give a constructive definition of well-founded models of logic programs as *iterated least partial models* or - equivalently - as *iterated fixed points* of the immediate consequence operator $\widehat{T}$ defined in Section 4.2. This generalizes the approach described in the previous Example 9.

Definition 12 [Prz89a, Prz90]. Let P be an arbitrary logic program and let J_0 be the empty interpretation $J_0 = \{\}$, in which all propositions are undefined (the interpretation J_0 represents our initial lack of information about the truth or falsity of propositions appearing in P).

Suppose now that J_n has already been defined for any $n < m$. If $m = n + 1$ then we define:

$$J_{n+1} = LPM(\frac{P}{J_n})$$

or, equivalently:

$$J_{n+1} = \Psi(J_n)$$

else, if m is a limit ordinal, we define:

$$J_m = \bigcup_{n<m} J_n . \square$$

As we explained in Example 9, at every consecutive successor step we compute the next iteration J_{n+1} of the interpretation J_n by computing the least partial model of the quotient program $\frac{P}{J_n}$ of P modulo J_n, thus possibly adding new information to our current body of knowledge encoded in J_n. At limit steps, we simply take the set-theoretic union of the previously constructed interpretations J_n, thus collecting together the previously deduced knowledge about P.

The following fundamental theorem proves that the above recursively defined sequence of interpretations J_n is always increasing (in the sense of set-theoretic inclusion) and thus always has a fixed point J_k with the property that $J_{k+1} = J_k$, or - equivalently - such that $J_k = \Psi(J_k) = LPM(\frac{P}{J_k})$. Moreover, this fixed point coincides with the *well-founded* model M_P of the program P, as originally defined in [VGRS90].

Theorem 13. (Well-founded Models as Iterated Least Partial Models) *([Prz89a, BNN90]) The sequence of interpretations J_n described in Definition 12 is always increasing (in the sense of set-theoretic inclusion) and thus always has a fixed point J_k with the property that*

$$J_{k+1} = J_k,$$

or - equivalently - such that

$$J_k = \Psi(J_k) = LPM(\frac{P}{J_k}).$$

Moreover, this fixed point J_k coincides with the well-founded *model M_P of the program P:*

$$M_P = J_k.$$

In addition, if P is a datalog *program, i.e., if it does not contain non-ground functional terms, or - equivalently - if the instantiated program is finite, then the above described iterative construction of the well-founded model of P stops after* finitely *many steps and is* quadratic *in the size of the program.* □

Remark. The characterization of well-founded models given above is *formally identical* to the characterization first given (without proof) in ([Prz89a]; Theorem 3.2). It is, however, stated here in a simpler and more natural way using the concepts of the least partial model and the quotient operator introduced earlier in [Prz91c]. Our formulation has been largely inspired by the paper written by Bonnier, Nilsson and Naslund [BNN90] where another formally identical, yet technically more complex, characterization of well-founded models as iterated least partial models was obtained. □

The above characterization shows that *well-founded models are simply iterated least partial models* of logic programs. Since least partial models are least fixed points of the immediate consequence operator $\widehat{T}$ defined in Section 4.2, well-founded models can also be viewed as *iterated least fixed points* of the immediate consequence operator $\widehat{T}$.

This constructive characterization of well-founded models is very important from the procedural point of view. Since it iterates the computation of *least partial models*, it constitutes a sequence of *linear computations.* Naturally, if the fixed point is attained at a transfinite step, only an *approximation* of the well-founded model can be obtained in finite time. We recall that the original definition of well-founded models given in [VGRS90] was not constructive and that there is *no* satisfactory constructive definition of stable models.

Definition 14. The *well-founded semantics* $WF(P)$ of a program P is defined as the set of all sentences satisfied in the unique well-founded model M_P of P. □

Example 10. As explained in Example 9, the well-founded model of the program P_8:

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

is $M_P = \{\neg a,\ b,\ \neg c\}$. We note that this program does not have any stable models. □

Example 11. Consider the program P_4 discussed in Section 2.1:

$$\begin{aligned} work &\leftarrow \neg tired \\ sleep &\leftarrow \neg work \\ tired &\leftarrow \neg sleep \\ angry &\leftarrow \neg paid, work \\ paid &\leftarrow \end{aligned}$$

Initially, we have no knowledge available about the program. Therefore our initial knowledge is best represented by the empty interpretation $J_0 = \{\}$, in which all propositions are undefined. The quotient $\frac{P_4}{J_0}$ of P_4 modulo J_0 is given by:

$$\begin{aligned} work &\leftarrow u \\ sleep &\leftarrow u \\ tired &\leftarrow u \\ angry &\leftarrow u\ , work \\ paid &\leftarrow \end{aligned}$$

and its least partial model $J_1 = \Psi(J_0) = LPM(\frac{P_4}{J_0})$ is the model $\{paid\}$, in which *paid* is true and the remaining propositions are undefined.

We repeat the procedure starting with the interpretation J_1 which now represents our current knowledge. The quotient $\frac{P_4}{J_1}$ of P_4 modulo J_1 is given by:

$$\begin{aligned} work &\leftarrow u \\ sleep &\leftarrow u \\ tired &\leftarrow u \\ angry &\leftarrow f\ , work \\ paid &\leftarrow \end{aligned}$$

and its least partial model $J_2 = \Psi(J_1) = LPM(\frac{P_4}{J_1})$ is the model $\{paid,\ \neg angry\}$, in which *paid* is true, *angry* is false and the remaining propositions are undefined.

If we repeat the above procedure starting with the interpretation J_2 which represents our current knowledge, we discover that the quotient $\frac{P_4}{J_2}$ of P_4 modulo J_2 is exactly the same as before and therefore its least partial model $\Psi(J_2) = LPM(\frac{P_4}{J_2})$ is J_2 itself. We have therefore reached a fixed point $\Psi(J_2) = J_2$, i.e., we can no longer add any new information to our knowledge about the program P_4, and therefore the model $J_2 = \{paid, \neg angry\}$ is the well-founded model M_{P_4} of P_4. We note that this program does not have any stable models. $\square$

Example 12. [GL88] Consider the program P_9 given by:

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

After instantiating, P_9 takes the form:

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

The quotient $\frac{P_9}{J_0}$ of P_9 modulo J_0 is given by:

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

and its least partial model $J_1 = \Psi(J_0) = LPM(\frac{P_9}{J_0})$ is the model

$$\{p(1,2),\ \neg p(1,1),\ \neg p(2,2),\ \neg p(2,1),\ \neg q(2)\}.$$

Now, we repeat the procedure starting with the interpretation J_1 which now represents our current knowledge. The quotient $\frac{P_9}{J_1}$ of P_9 modulo J_1 is given by:

$$\begin{array}{ll} p(1,2) \leftarrow & \\ q(1) & \leftarrow p(1,2), t \\ q(1) & \leftarrow p(1,1), u \\ q(2) & \leftarrow p(2,2), t \\ q(2) & \leftarrow p(2,1), u\ . \end{array}$$

and its least partial model $J_2 = \Psi(J_1) = LPM(\frac{P_9}{J_1})$ is the model

$$\{p(1,2),\ \neg p(1,1),\ \neg p(2,2),\ \neg p(2,1),\ q(1),\ \neg q(2)\}.$$

If we repeat the above procedure starting with the interpretation J_2 which represents our current knowledge, we discover that the quotient $\frac{P_9}{J_2}$ of P_9 modulo J_2 is:

$$\begin{array}{ll} p(1,2) \leftarrow & \\ q(1) & \leftarrow p(1,2), t \\ q(2) & \leftarrow p(2,2), t \end{array}$$

and the least partial model $LPM(\frac{P_9}{J_2})$ of $\frac{P_9}{J_2}$ is J_2 itself. We have therefore reached a fixed point $J_2 = \Psi(J_2)$ and therefore the model

$$J_2 = \{p(1,2),\ \neg p(1,1),\ \neg p(2,2),\ \neg p(2,1),\ q(1),\ \neg q(2)\}$$

is the well-founded model M_{P_9} of P_9. The program is not locally stratified, but since its well-founded model is total, by Theorem 16, it coincides with the unique stable model of P_9. □

Example 13. Finally, consider the program P_7 discussed in Example 4:

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

Again, initially, we have no knowledge available about the program. Therefore our initial knowledge is best represented by the empty interpretation $J_0 = \{\}$. The quotient $\frac{P_7}{J_0}$ of P_7 modulo J_0 is given by:

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

and its least partial model $J_1 = \Psi(J_0) = LPM(\frac{P_7}{J_0})$ is again the empty interpretation $J_0 = \{\}$. Since we have reached a fixed point, the well-founded model M_{P_7} of P_7 is the empty model in which all propositions are undefined. As we pointed out in Example 4 this model is different from the unique stable model of P_7 and appears to be more intuitive. □

The following theorem follows immediately from the characterization of well-founded models as iterated least partial models:

Theorem 15. *An interpretation M is the well-founded model of a logic program P if and only if M is the smallest (in the sense of inclusion) fixed point of the operator Ψ.* □

Thus well-founded models can also be viewed as smallest fixed points of a natural minimal model operator $\Psi(I) = LPM(\frac{P}{I})$. In the next section we specifically study the class of all fixed points of Ψ.

Finally, we recall here the basic result proved in [VGRS90] relating the well-founded semantics to the perfect model semantics.

Theorem 16. *[VGRS90] For all (locally) stratified programs P the well-founded model M_P of P coincides with the perfect model of P.* □

4.5 Stationary Models

In this section we define *stationary* or *partial stable models* originally introduced in [Prz91c]. The class of stationary (or partial stable) models includes the class of stable models defined in [GL88]. Stationary models are defined as fixed points of the operator Ψ described in the previous section:

Definition 17 Stationary Models. [Prz91c] A partial interpretation M of a logic program P is called a *stationary* or a *partial stable* model of P if it is a fixed point of the least model operator Ψ:

$$M = \Psi(M).$$

Thus M is a stationary model of P if and only if M is the least partial model of $\frac{P}{M}$:

$$M = LPM(\frac{P}{M}).$$

The *stationary* or *partial stable* semantics $STAT(P) = PSTABLE(P)$ of a program P is determined by the set of all stationary models of P, i.e., a sentence F is true in $STAT(P)$ if and only if it is true in all stationary models of P. □

It is easy to see that any stationary model of P is always a *minimal* partial model of P with respect to the truth ordering $\preceq$. Moreover, stationary models extend the class of stable models:

Proposition 18. *[Prz91c] For any logic program P, stable models coincide with* total *stationary models.* □

In general, a logic program may have more than one stationary model. For example, one can easily verify that the program:

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

has three stationary models, two of which are total. In one of them a is true and b false, in the other b is true and a false and in the third both a and b are undefined. When originally defining the stable model semantics, Gelfond and Lifschitz considered only those programs which have a unique (total) stable model. We do not make any such assumption.

4.6 Well-Founded Model Coincides with the Smallest Stationary Model

It follows immediately from Theorem 15 that well-founded models always coincide with smallest stationary (or partial stable) models.

Corollary 19. *[Prz91c] The well-founded model of an arbitrary logic program P always coincides with the smallest (in the sense of inclusion) stationary model of P.* □

In other words, the well-founded model M_P is the smallest stationary model of P, in the sense that, of all stationary models, M_P contains the least number of true or false facts, and, thus, the largest set of undefined facts. We can say, borrowing from Horty and Thomasson's inheritance network terminology [HTT87], that the well founded model is the most *skeptical* stationary model or possible world for P. For example, if P is given by $a \leftarrow \neg b,\ b \leftarrow \neg a$, then, as we have seen before, P has three stationary 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. Similarly, if P is the program from Example 4, then the 'most undefined' stationary model of P is well-founded. This can be explained by saying that the well-founded semantics 'believes' only in those things which hold in all possible worlds (i.e., stationary or partial stable models) of the program.

Observe that although the above characterization of well founded models as smallest stationary models is mathematically elegant it does not provide any *constructive* way of finding such models. Constructive definition of well-founded models was given in the previous section.

Since the well-founded model is a set-theoretic intersection of all stationary models, we immediately obtain:

Corollary 20. *[VGRS90] If a program P has a* total *well-founded model M_P then it also has a unique stable model and the two coincide.* □

Since the stationary or partial stable semantics $STAT(P) = PSTABLE(P)$ of a program P is determined by the class of all stationary or partial stable models and since the well-founded model is the smallest stationary model, it immediately follows that the well-founded semantics of any program P always coincides with the stationary semantics of P.

Theorem 21. *The well-founded semantics of an arbitrary logic program P coincides with the stationary (or partial stable) semantics of P in the sense that for any sentence F (not containing the connective "$\leftarrow$"):*

$$WF(P) \models F \quad \equiv \quad STAT(P) \models F \quad \equiv \quad PSTABLE(P) \models F. \Box$$

The above result immediately implies that the well-founded semantics is, in general, weaker than the stable semantics $STABLE(P)$ which is based only on (total) stable models.

4.7 Dynamic Stratification

We will now use the iterated least partial model definition of the well-founded model M_P given in the Section 4.4 to introduce the *dynamic stratification* of an *arbitrary* logic program P. This will show that well-founded models share the important property of perfect models, namely the fact that they are iterated least models of logic programs with respect to their stratification.

Recall that according to Definition 12 the well-founded model M_P is defined as the fixed point J_k, $k \geq 0$, of a strictly increasing sequence of interpretations (iterated least partial models) J_n:

$$J_0 \subset \ldots \subset J_n \subset \ldots \subset J_k = J_{k+1},$$

starting with the empty interpretation $J_0 = \{\}$. The interpretations J_n represent our *increasing knowledge* about the truth values of ground atoms in the (instantiated) program P, i.e., they describe the set of ground atoms currently known to be either *true* or *false* in J_n.

The *dynamic stratification* $\{D_n\}$ *of* P is a decomposition of the *Herbrand base* H_P of P, i.e., a decomposition of the set of all ground atoms, into disjoint sets D_n, $1 \leq n \leq k+1$, which are called *dynamic strata.* For any $n \leq k$, the n-th stratum D_n is defined as the set of those ground atoms, whose truth or falsity has been *precisely determined at the level* n. The last stratum D_{k+1} contains all the remaining ground atoms, i.e., those ground atoms which are undefined in $M_P = J_k$. Below we give a formal definition:

Definition 22. Let P be an arbitrary logic program and let J_n, $0 \leq k$ be the strictly increasing sequence of interpretations (iterated least partial models) defining the well-founded model $M_P = J_k$ of P.

For any $1 \leq n \leq k$, we define the *n-th dynamic stratum* D_n of P to be the set of those ground atoms which are either true or false in the interpretation J_n, but are undefined in all previous interpretations J_m, with $m < n$.

The last, $k+1$*-st dynamic stratum* D_{k+1} is defined as the set of all the remaining ground atoms, i.e., those ground atoms which are undefined in the well-founded model $M_P = J_k$. □

Clearly, the well-founded model M_P is *total* if and only if the last stratum D_{k+1} is empty.

Example 14. **(see Example 9)** According to the discussion in Example 9, the dynamic stratification of the program P_8:

$$\begin{array}{l} b \leftarrow \sim a \\ c \leftarrow \sim b, p \\ p \leftarrow \sim p \end{array}$$

is given by:

$$\begin{array}{l} D_1 = \{a\} \\ D_2 = \{b\} \\ D_3 = \{c\} \\ D_4 = \{p\}. \end{array}$$

(see Example 11) The dynamic stratification of the program P_4:

$$\begin{array}{ll} work & \leftarrow \sim tired \\ sleep & \leftarrow \sim work \\ tired & \leftarrow \sim sleep \\ angry & \leftarrow \sim paid, work \\ paid & \leftarrow \end{array}$$

is given by:

$$\begin{aligned}
&D_1 = \{paid\}\\
&D_2 = \{angry\}\\
&D_3 = \{sleep,\ work,\ tired\}.
\end{aligned}$$

(see Example 12) The dynamic stratification of the program P_9:

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

is given by:

$$\begin{aligned}
&D_1 = \{p(1,2),\ p(1,1),\ p(2,1),\ p(2,2,),\ q(2)\}\\
&D_2 = \{q(1)\}\\
&D_3 = \{\}
\end{aligned}$$

In this case the last stratum D_3 is empty, because M_P is total.
(see Example 13) The dynamic stratification of the program P_7:

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

is given by:

$$D_1 = \{a,\ b,\ p\}.$$

In this case the last stratum D_1 contains all ground atoms, because M_P is empty. □

When restricted to the class of (locally) stratified programs, dynamic stratification is *essentially* the same as standard stratifications, but the two notions, in general, do *not* coincide. The causes of this discrepancy can be easily explained.

First of all, dynamic stratification, as opposed to standard stratifications, is *uniquely defined* for any program P. Secondly, standard stratifications are determined *statically*, based on the syntactical form of the program or – more precisely – on its dependency graph [ABW88, VG89b]. Therefore they do not depend in any way on the truth or falsity of the atoms appearing in the graph. This makes them very simple and natural to use, but, at the same time, *limits* their applicability to a relatively narrow class of programs.

On the other hand, dynamic stratification – as the name indicates – is generated *dynamically*. This means that, in the process of stratification, irrelevant dependencies between atoms (i.e. those that can never be satisfied) are eliminated. As a result, we obtain a stratification that may be considered *preferable* to the standard ones in the sense described by the following theorem.

Theorem 23. *[Prz89a] Suppose that P is a (locally) stratified program. Let $\{S_n\}$ be any of its standard stratifications and let $\{D_n\}$ be its dynamic stratification. For every ground atom A,*

$$(A \in D_m \ \wedge \ A \in S_n) \Rightarrow m \leq n. \Box$$

Appendix.

In other words, the levels that the dynamic stratification assigns to ground atoms are less than or equal to those assigned by standard stratifications. One can say that dynamic stratification is the *tightest* of all stratifications. The idea of dynamic stratification originated in [PP90b].

To illustrate the phenomenon discussed above, consider the following example.

Example 15. Let P_{10} be given by:

$$\begin{array}{l} a \\ b \leftarrow \neg a \\ c \leftarrow \neg b, \neg a. \end{array}$$

This program is stratified and its (standard) stratification is given by: $S_1 = \{a\}$, $S_2 = \{b\}$, $S_3 = \{c\}$. On the other hand, we have:

$$\begin{array}{ll} J_1 = \{a\}; & D_1 = \{a\}; \\ J_2 = \{a, \ \neg b, \ \neg c\}; & D_2 = \{b, c\}; \\ J_3 = J_2 = M_P; & D_3 = \{\}. \end{array}$$

The last stratum D_3 is empty because the well-founded (perfect) model $M_P = \{a, \ \neg b, \ \neg c\}$ of the program P_{10} is total. $\Box$

Consequently, dynamic stratification requires only two strata instead of three required by the standard stratification. This is caused by the fact that, in the process of constructing the second dynamic stratum, D_2, we observe that one of the premises of the third clause, namely $\neg a$, is already known to be false. Therefore, the third clause is at this point irrelevant. Consequently, c does not need to be put into a higher stratum, because it *does not* really depend negatively on b.

Remark. It is easy to see that – if one is to ignore undefined atoms – the differences between dynamic stratification and static stratifications are caused exclusively by the existence of some (ground instances) of program clauses, which affect the dependency graph and thus influence standard stratifications, but are irrelevant from the point of view of dynamic stratification, because they contain some premises which are false in the well-founded model. The third clause in the previous example illustrates this point. If it were replaced by the clause:

$$c \leftarrow \neg a$$

then the resulting stratified program P'_{10} would be semantically equivalent to the original program P_{10}, and the (dynamic or static) stratification of P'_{10} would coincide with the dynamic stratification of the original program P_{10}.

One can show that a similar construction can be performed for an arbitrary (instantiated) logic program P, whose well-founded model is *total*, leading to a semantically equivalent stratified program P', with the property that the dynamic stratification of P coincides with the static stratification of P'. One may therefore view programs with total well-founded models as *'stratified programs in disguise'*, because they only *'pretend'* not to be stratified, by including some irrelevant clauses which destroy their standard stratification.

4.8 Non-Herbrand Models

Throughout the paper, with the exception of Clark's semantics and its extension, due to Fitting and 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 identified as the *universal query problem* in [Prz89b].

Suppose that our program P consists of a trivial clause $p(a)$. The program is positive and has only one 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 argued in [Prz89b] – 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

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

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

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

etc., signifying that

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

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 only know 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 [Prz89b]) is to *extend the definitions of intended models* to include *non-Herbrand* models, thus leading to the definitions of non-Herbrand perfect models (respectively, 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, *not necessarily Herbrand*, perfect models (respectively, 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 many contexts may seem most natural.
   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 [Prz89b] 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 non-Herbrand models see [GPP88, Prz89b].

4.9 Relationship to Non-Monotonic Formalisms

Non-monotonic reasoning and logic programming are closely related. On the one hand, the non-monotonic character of the "negation by default" operator used in logic programming allows us to view logic programs as special, fairly simple and yet quite expressive, non-monotonic theories. Consequently, the problem of finding a suitable semantics for logic programs can be viewed as the problem of formalizing the type of non-monotonic reasoning used in logic programming.

On the other hand, logic programs constitute an important class of non-monotonic theories which, due to their relative simplicity, can be conveniently used to test the behavior of various non-monotonic formalisms and can also give rise to new non-monotonic formalisms based on the ideas originating in logic programming. Finally, logic programs, equipped with a suitable semantics, can be used as relatively efficient inference engines for non-monotonic reasoning.

In spite of the close relationship between non-monotonic reasoning and logic programming, 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 2). Clark's formalism is not sufficiently general to be applied beyond the realm of logic programming and therefore it does not play a significant role in non-monotonic reasoning.

The situation has changed significantly with the introduction of stratified logic programs and the perfect model semantics (see Section 3.3). 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 [Prz89b, Lif88];
- Reiter's default theory [BF87, MT89];
- Moore's autoepistemic logic [Gel87];
- Reiter's CWA [GPP89].

The stable model semantics [GL88], discussed in Section 3.4, extends the perfect model semantics and is also closely related to non-monotonic formalisms. Gelfond proved in [Gel87] that there is a one-to one correspondence between stable models of a logic program P and stable autoepistemic expansions of its translation $\widehat{P}$ into Moore's autoepistemic logic [Moo85]. Bidoit and Froidevaux showed in [BF91] that there also exists a one-to one correspondence between stable models of a logic program P and default extensions of its translation $\overline{P}$ into Reiter's default theory [Rei80].

The original definition of well-founded semantics given in [VGRS90] did not seem to provide any clues as far as its relationship to non-monotonic formalisms is concerned. However, Przymusinski has shown in [Prz91d] 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.

Recently, Przymusinski [Prz94a, Prz94b] introduced the notion of a *static autoepistemic expansion* in which beliefs are based on McCarthy's *Circumscription*

CIRC [McC80], or - equivalently - on Minker's *Generalized Closed World Assumption* $GCWA$ [Min82], rather than on Reiter's CWA, as in Moore's original autoepistemic expansions. He proved that there is a one-to one correspondence between stationary (or partial stable) models of a logic program P and static autoepistemic expansions of its translation $\widehat{P}$ into autoepistemic logic. In particular, there is a one-to one correspondence between the well-founded model of a logic program P and the least static autoepistemic expansion of $\widehat{P}$. This result shows that *well-founded models*, and, more generally, *stationary models* of logic programs can also be defined in terms of autoepistemic logic.

Similarly, Przymusinska and Przymusinski [PP94] introduced the notion of a *stationary default extension* of Reiter's default theories and showed that there is a one-to one correspondence between stationary (or partial stable) models of a logic program P and stationary default extensions of its translation $\overline{P}$ into default theory. In particular, there is a one-to one correspondence between the well-founded model of a logic program P and the least stationary default extension of $\overline{P}$. This result shows that stationary models can as well be defined in the language of default theory [LY91, BS91].

4.10 Procedural Semantics: SLS-resolution

One of the important strengths of the well-founded semantics is the existence of a sound and complete[10] querry evaluation procedure naturally extending the well-known $SLDNF-resolution$. It is called *SLS-resolution* and was originally defined in [Prz89a, Prz89b] (see also [Ros89]). The definition presented here is slightly different, however, as it does not require the advance knowledge of the stratification of the program.

Suppose that P is any logic program. By a *goal* G we mean a headless clause $\leftarrow L_1$,..., L_k, where k$\geq$0 and L_i's are literals. We also write, $G = \leftarrow Q$, where Q = L_1 ,..., L_k is called a *query.*

Let us choose an arbitrary *computation rule* R (see [Llo87]), i.e., a rule that selects exactly one literal from any non-empty goal. We will define the SLS-tree, $SLS(G)$, for a goal G by constructing a sequence:

$$\{SLS_\beta(G)\}_{\beta\geq 0}, \quad SLS_\alpha(G) \preceq SLS_\beta(G), \quad \text{for} \quad \alpha < \beta,$$

of SLS-trees of rank β, with $SLS_\alpha(G)$ being a subtree of $SLS_\beta(G)$, for $\alpha < \beta$, and with $SLS(G)$ defined as the largest element (or the union) of the sequence.

Some of the *leaves* of the so constructed SLS-trees SLS_β will be labeled as *success*, *failure* and *flounder* leaves, respectively. A leaf which is not labeled will be called a *non-labeled* leaf. We will use the following definition:

Definition 24. (Successful, Failed and Floundered SLS-trees.) An SLS-tree for a goal G is *successful* if it has a *successful derivation*, i.e., a derivation ending in a success leaf. It is *failed* if all of its branches are either infinite or end in a failure leaf. It is *floundered* if it contains a flounder leaf and is not successful. □

[10] For non-floundering querries.

We now define SLS-trees of rank β. The SLS-tree, $SLS_0(G)$, of rank $\beta = 0$ consists only of the non-labeled root node G. Suppose now that $\beta > 0$ and assume that SLS-trees, $SLS_\alpha(G)$, of rank α have been already defined for all $0 \leq \alpha < \beta$ and for all goals G.

If β is a limit ordinal then the SLS-tree of rank β is defined as the union of all the previously constructed trees $SLS_\alpha(G)$, for $\alpha < \beta$:

$$SLS_\beta(G) = \bigcup_{\alpha<\beta} SLS_\alpha(G).$$

Otherwise, $\beta = \alpha + 1$ and we define the SLS-tree $SLS_{\alpha+1}(G)$ by extending the previously constructed SLS-tree of rank α by possibly adding some descendants to (or by labeling) some of the non-labeled leaves of $SLS_\alpha(G)$:

Definition 25. (SLS-trees of rank $\alpha + 1$) The SLS-tree, $SLS_{\alpha+1}(G)$, of rank $\alpha + 1$ consists of all nodes (and labels) of the previously constructed SLS-tree, $SLS_\alpha(G)$, of rank α together with all nodes (and labels) obtained by recursively applying the following resolution rules to all *non-labeled leaves* of $SLS_\alpha(G)$ and to their *descendants.*

Let $H = \leftarrow L_1, \ldots, L_k$ be an arbitrary non-labeled leaf of $SLS_\alpha(G)$ (or a descendent thereof) and let L be the literal selected from H by the computation rule R. The immediate descendants (successors) of H in the tree $SLS_{\alpha+1}(G)$ are defined as follows:

(i) If H is *empty* then it has no descendants and is a *success leaf.*

(ii) If $L = A$ is a positive literal then the immediate descendants of H are – as usual – all goals K that can be obtained from the goal H by resolving H with (a variant of) one of the program clauses upon the atom A, using most general unifiers.

If there are no such K's, then H has no descendants and is a *failure leaf*;

(iii) If $L = \neg A$ is negative then we consider four cases:

- If A is ground and the SLS-tree $SLS_\alpha(\leftarrow A)$ is *failed* then H has precisely one descendent, namely, the node $K = H - \{L\}$ obtained by removing L from H;
- If A is ground and the SLS-tree $SLS_\alpha(\leftarrow A)$ is successful then the node H has no descendants and is called a *failure leaf*;
- If A is not ground or if the SLS-tree $SLS_\alpha(\leftarrow A)$ is floundered then the node H has no descendants and is a *flounder leaf*;
- Otherwise, we mark the literal L in H as *skipped* and use the computation rule[11] R to select, if possible, a new literal L' and subsequently apply the resolution steps (ii) and (iii) to the goal H, with L' now acting as the literal selected by R.

  If all literals in the goal H were already marked as skipped then the node H has no descendants in $SLS_{\alpha+1}(G)$ (and is not labeled). □

[11] Applied to the goal H from which all literals marked as skipped were removed.

It follows immediately from the definition that for any goal G the sequence $\{SLS_\beta(G)\}_{\beta \geq 0}$ of SLS-trees of rank β is non-decreasing, i.e.,

$$SLS_\alpha(G) \preceq SLS_\beta(G), \quad \text{for} \quad \alpha < \beta,$$

and therefore, due to the countability of the language, it must have a fixed point, i.e., there must exist a δ such that:

$$SLS_\delta(G) = SLS_{\delta+1}(G).$$

We define the SLS-tree $SLS(G)$ for the goal G as its SLS-tree of rank δ:

$$SLS(G) = SLS_\delta(G).$$

We define SLS-computed answer substitutions in the usual way:

Definition 26. (SLS-computed answer substitutions) If the SLS-tree for a goal G has a successful derivation then the accumulated substitution θ, restricted to the variables in G, is called the *SLS-computed answer substitution* for G. □

The following result originally stated in [Prz89a] (see also [Ros89]) shows that SLS-resolution is *sound and complete* (for non-floundering queries) w.r.t. the well-founded semantics:

Theorem 27 Soundness and Completeness of SLS-resolution. *[PPS92] Suppose that P is a logic program, R is a computation rule and $G = \leftarrow Q$ is a goal. If the SLS-tree for G is non-floundered then:*

(i) $WF(P) \models \exists(Q)$ *iff the SLS-tree for* $\leftarrow Q$ *is successful;*
(ii) $WF(P) \models \forall(Q\theta)$ *iff there exists an SLS-computed answer substitution for* $\leftarrow Q$ *more general than the substitution* θ*;*
(iii) $WF(P) \models \neg\exists(Q)$ *iff the SLS-tree for* $\leftarrow Q$ *is failed.* □

5 Conclusion

In this paper we discussed the class of stationary or partial stable models of normal logic programs. This important class of models includes all (total) stable models and the well-founded model is always its smallest member. They have several natural fixed-point definitions and can be equivalently obtained as expansions or extensions of suitable autoepistemic or default theories. By taking a particular subclass of this class of models one can obtain different semantics of logic programs, including the stable semantics and the well-founded semantics. Stationary models can be also naturally extended to the class of all disjunctive logic programs (see [Prz94b, Prz91b]. These features of stationary models designate them as an important class of models with applications reaching far beyond the realm of logic programming.

References

[AB90] K. R. Apt and M. Bezem. Acyclic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 617–633. The MIT Press, 1990.

[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.

[BNN90] S. Bonnier, U. Nilsson, and T. Naslund. A simple fixed point characterization of three-valued stable model semantics. Research report, University of Linkoping, 1990.

[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.

[BS91] C. Baral and V.S. Subrahmanian. Dualities between alternative semantics for logic programming and non-monotonic reasoning. In A. Nerode, W. Marek, and V.S. Subrahmanian, editors, *Proceedings of the First International Workshop on Logic Programming and Non-monotonic Reasoning, Washington, D.C., July 1991*, pages 69–86, Cambridge, Mass., 1991. MIT Press.

[Cav89] L. Cavedon. Consistency and completeness properties for logic programs. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Logic Programming Conference, Lisbon, Portugal*, pages 571–584, Cambridge, Mass., 1989. Association for Logic Programming, MIT Press.

[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.

[CW92] W. Chen and D. S. Warren. A practical approach to computing the well-founded semantics. Research report, SUNY at Stony Brook, 1992.

[Dix91] J. Dix. Classifying semantics of logic programs. In A. Nerode, W. Marek, and V.S. Subrahmanian, editors, *Proceedings of the First International Workshop on Logic Programming and Non-monotonic Reasoning, Washington, D.C., July 1991*, pages 166–180, Cambridge, Mass., 1991. MIT Press.

[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.

[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.

[HTT87] J. Horty, R. Thomason, and D. Touretzky. A skeptical theory of inheritance in non-monotonic semantic nets. In *Proceedings AAAI-87*, Los Altos, CA, 1987. American Association for Artificial Intelligence, Morgan Kaufmann.

[KS89] H. A. Kautz and B. Selman. Hard problems for simple default logics. 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 189–197. Morgan Kaufmann, 1989.

[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.

[LY91] L. Li and J.H. You. Making default inferences from logic programs. *Journal of Computational Intelligence*, 7:142–153, 1991. In print.

[Mak88] D. Makinson. General theory of cumulative inference. In *Proceedings of the Second Workshop on Non-monotonic Reasoning, Munich, July 1988*, pages 1–18. Springer Verlag, 1988.

[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.

[Moo85] R.C. Moore. Semantic considerations on non-monotonic logic. *Journal of Artificial Intelligence*, 25:75–94, 1985.

[MT88] W. Marek and M. Truszczynski. Autoepistemic logic. Research report, University of Kentucky, University of Kentucky, 1988.

[MT89] W. Marek and M. Truszczynski. Relating autoepistemic and default logics. 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 189–197. Morgan Kaufmann, 1989.

[PP90a] H. Przymusinska and T. C. Przymusinski. Semantic issues in deductive databases and logic programs. In R. Banerji, editor, *Formal Techniques in Artificial Intelligence*, pages 321–367. North-Holland, Amsterdam, 1990.

[PP90b] H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990.

[PP94] H. Przymusinska and T. C. Przymusinski. Stationary default extensions. *Fundamenta Informaticae*, 20:(In print.), 1994. Special issue devoted to the Fourth Workshop on Non-monotonic Reasoning, Plymouth, Vermont, 1992.

[PPS92] H. Przymusinska, T. Przymusinski, and H. Seki. Soundness and completeness of partial deductions for well-founded semantics. In A. Voronkov, editor, *Logic Programming and Automated Reasoning, St. Petersburg, Russia, July 1992. (Lecture Notes in Artificial Intelligence, vol. 624)*, pages 1–12. Springer Verlag, 1992.

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

[Prz90] T. C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13(4):445–464, 1990.

[Prz91a] T. C. Przymusinski. Autoepistemic logics of closed beliefs and logic programming. In A. Nerode, W. Marek, and V.S. Subrahmanian, editors, *Proceedings of the First International Workshop on Logic Programming and Non-monotonic Reasoning, Washington, D.C., July 1991*, pages 3–20, Cambridge, Mass., 1991. MIT Press.

[Prz91b] T. C. Przymusinski. Semantics of disjunctive logic programs and deductive databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases DOOD'91*, pages 85–107, Munich, Germany, 1991. Springer Verlag.

[Prz91c] T. C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing Journal*, 9:401–424, 1991. (Extended abstract appeared in: Extended stable semantics for normal and disjunctive logic programs. *Proceedings of the 7-th International Logic Programming Conference, Jerusalem*, pages 459–477, 1990. MIT Press.).

[Prz91d] T. C. Przymusinski. Three-valued non-monotonic formalisms and semantics of logic programs. *Journal of Artificial Intelligence*, 49:309–343, 1991. (Extended abstract appeared in: Three-valued non-monotonic formalisms and logic programming. *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89), Toronto, Canada*, pages 341–348, Morgan Kaufmann, 1989.).

[Prz94a] T. C. Przymusinski. A knowledge representation framework based on autoepistemic logic of minimal beliefs. In *Proceedings of the Twelfth National*

Conference on Artificial Intelligence, AAAI-94, Seattle, Washington, August 1994, page (in print), Los Altos, CA, 1994. American Association for Artificial Intelligence, Morgan Kaufmann.

[Prz94b] T. C. Przymusinski. Static semantics for normal and disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 1994. (in print).

[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.

[Rei80] R. Reiter. A logic for default theory. *Journal of Artificial Intelligence*, 13:81–132, 1980.

[Ros89] K. Ross. A procedural semantics for well founded negation in logic programs. In *Proceedings of the Eighth Symposium on Principles of Database Systems*, pages 22–33. ACM SIGACT-SIGMOD, 1989.

[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.

[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.

[War89] David S. Warren. The XWAM: A machine that integrates Prolog and deductive database query evaluation. Technical report #25, SUNY at Stony Brook, 1989.