Prolegomena to Logic Programming for Non–Monotonic Reasoning

Jürgen Dix[1] and Luís Moniz Pereira[2] and Teodor Przymusinski[3]

[1] Dept. Computer Science, University of Koblenz
Rheinau 1, D-56075 Koblenz, Germany
dix@mailhost.uni-koblenz.de

[2] Dept. Computer Science and CENTRIA, Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal
lmp@di.fct.unl.pt

[3] Dept. Computer Science, University of California at Riverside
Riverside, CA 92521, USA
teodor@cs.ucr.edu

Abstract. The present prolegomena consist, as all indeed do, in a critical discussion serving to introduce and interpret the extended works that follow in this book. As a result, the book is not a mere collection of excellent papers in their own specialty, but provides also the basics of the motivation, background history, important themes, bridges to other areas, and a common technical platform of the principal formalisms and approaches, augmented with examples.

In the introduction we whet the reader's interest in the field of logic programming and non–monotonic reasoning with the promises it offers and with its outstanding problems too. There follows a brief historical background to logic programming, from its inception to actuality, and its relationship to non–monotonic formalisms, stressing its semantical and procedural aspects.

The next couple of sections provide motivating examples and an overview of the main semantics paradigms for normal programs (stable models and well–founded) and for extended logic programs (answer–sets, e–answer–sets, Nelson's strong negation, and well–founded semantics with pseudo and with explicit negation).

A subequent section is devoted to disjunctive logic programs and its various semantical proposals.

To conclude, a final section on implementation gives pointers to available systems and their sites.

We leave out important concerns, such as paraconsistent semantics, contradiction removal, and updates. Hopefully they will be included in the next book in this series. But an extensive set of references allows the reader to delve into the specialized literature.

For other recent relevant complementary overviews in this area we refer to [AP96,BDK97,BD96b,Min96,Dix95c].

1 Introduction

One of the major reasons for the success story (if one is really willing to call it a success story) of human beings on this planet is our ability to invent tools that help us improve our — otherwise often quite limited — capabilities. The invention of machines that are able to do interesting things, like transporting people from one place to the other (even through the air), sending moving pictures and sounds around the globe, bringing our email to the right person, and the like, is one of the cornerstones of our culture and determines to a great degree our everyday life.

Among the most challenging tools one can think of are machines that are able to handle knowledge adequately. Building smart machines is at the heart of Artificial Intelligence (AI). Since such machines will need tremendous amounts of knowledge to work properly, even in very limited environments, the investigation of techniques for representing knowledge and reasoning is highly important.

In the early days of AI it was still believed that modeling general purpose problem solving capabilites, as in Newell and Simon's famous GPS (General Problem Solver) program, would be sufficient to generate intelligent behaviour. This hypothesis, however, turned out to be overly optimistic. At the end of the sixties people realized that an approach using available knowledge about narrow domains was much more fruitful. This led to the expert systems boom which produced many useful application systems, expert system building tools, and expert system companies. Many of the systems are still in use and save companies millions of dollars per year[1].

Nevertheless, the simple knowledge representation and reasoning methods underlying the early expert systems soon turned out to be insufficient. Most of the systems were built based on simple rule languages, often enhanced with ad hoc approaches to model uncertainty. It became apparent that more advanced methods to handle incompleteness, defeasible reasoning, uncertainty, causality and the like were needed.

This insight led to a tremendous increase of research on the foundations of knowledge representation and reasoning. Theoretical research in this area has blossomed in recent years. Many advances have been made and important results were obtained. The technical quality of this work is often impressive.

On the other hand, most of these advanced techniques have had surprisingly little influence on practical applications so far. To a certain degree this is understandable since theoretical foundations had to be laid first and pioneering work was needed. However, if we do not want research in knowledge representation to remain a theoreticians' game, more emphasis on computability and applicability seems to be needed. We strongly believe that the kind of research presented in this book, that is research aiming at interesting combinations of ideas from logic programming and non–monotonic reasoning, and its implementation, provides an important step into this direction.

[1] We refer the interested reader to the recent book [RN95] which gives a very detailed and nice exposition of what has been done in AI since its very beginning until today.

Research in the area of logic programming and non-monotonic reasoning makes a significant contribution not only towards the better understanding of relations existing between various formalizations of non-monotonic reasoning, and, hopefully, towards the eventual discovery of deeper underlying principles of non-monotonic reasoning and logic programming, but, perhaps more importantly, towards the eventual development of relatively efficient inference engines based on the techniques originating in logic programming. Needless to say, the problem of finding efficient inference mechanisms, capable of modelling human common-sense reasoning, is one of the major research and implementation problems in AI. Moreover, by incorporating the recent theoretical results, the next qualitative leap in logic programming systems is well under way.

In fact, during the last decade a truly impressive body of knowledge has been accumulated, providing us with a better understanding of semantic issues in logic programming and the nature of its relationship to non-monotonic formalisms. New, significantly improved, semantics for logic programs have been introduced and thoroughly investigated and their close relationship to major non-monotonic formalisms has been established. A number of fundamental results describing ways in which non-monotonic theories can be translated into logic programs have been obtained. Entirely new computational mechanisms for the new, greatly improved, semantics have been introduced and several very promising implementations of logic programming systems based on these semantics have been developed.

In spite of these unquestionable successes some major problems remain:

1. Standard logic programs are not sufficiently expressive for the representation of large classes of knowledge bases. In particular, the inability of logic programs to deal with disjunctive information proved to be a major obstacle to using them effectively in various knowledge domains, in particular, it constitutes a major obstacle to using logic programming as a declarative specification language for software engineering. The problem of extending of the logic programming paradigm to the class of disjunctive programs and deductive databases turned out to be a difficult one, as evidenced by a large number of papers, the book [LMR92] and several workshops ([Wol94,DLMW96]) devoted to this issue. It is well known that disjunctive programs are significantly more expressive than normal programs.
2. In recent years, several authors have underscored the importance of extending logic programming by introducing different types of negation (explicit, strong or classical negation) which proved essential for knowledge representation and are indispensable for the effective use of integrity constraints. This important enhancement of logic programming immediately leads, however, to the new basic problem of how to deal with contradiction in such extended logic programs. The proper definition of declarative and procedural semantics for contradiction removal is particularly crucial in applications of logic programming to diagnosis, database updates, and declarative debugging.
3. The existing logic programming systems suffer from an important procedural limitation of being unable to deal with non-ground negative queries.

This well-known phenomenon, known as the *floundering problem*, severely restricts the usefulness of logic programs for knowledge representation. Several major attempts have been made recently to deal with this problem but much more work is still needed.

4. In spite of the unquestionable progress obtained during the last decade or so, insufficient attention has been paid to the study of various applications of the new logic programming paradigm and to the actual testing and experimentation with the new systems in various application domains. Extensive experimentation will allow us to better understand and evaluate their potential value for knowledge representation, at the same time exposing their possible weaknesses and inadequacies.
5. While several promising implementations of logic programming systems have been developed recently, much more effort has to be devoted to this issue during the next several years if we are to come up with a truly practical and efficient alternative to the currently existing logic programming paradigm. These efforts must include implementations which are capable, among others, to deal with disjunctive information, contradiction removal and constructive negation. More work is also needed towards the development of object-oriented logic programming systems.

In short, the advent of low cost multiprocessor machines, impressive research progress of the past decade as well as significant advances in logic programming implementation techniques now provide us with a great opportunity to bring to fruition computationally efficient implementations of extended logic programming. The resulting programming system must not only ensure the increased expressivity and declarative clarity of the new paradigm of logic programming but it should also be suitable to serve as an inference engine for other non-monotonic reasoning formalisms and deductive databases and as a specification language for software engineering.

This book is the successor of [DPP95] and the next in a series addressing such issues. To facilitate access to the book, and to this research area, we decided to provide here an introduction and motivation for (extended and disjunctive) logic programming, which is commonly used as a basis for knowledge representation and non–monotonic reasoning, and applications. Other more specific developments, such as belief revision or updates, are deliberately left out. The reader is referred to the collection of papers in this book and the references therein for such topics.

2 Brief historical logic programming background

Computational Logic arose from the work begun by logicians since the midfifties. The desire to impart the computer with the ability to reason logically led to the development of automated theorem proving, which took up the promise of giving logic to artificial intelligence. This approach was fostered in the 1970's by Colmerauer et al. [CKPR73] and Kowalski [Kow74,Kow79] as Logic Programming (and brought on the definition and implementation of *Prolog* [CKPR73]).

It introduced to computer science the important concept of *declarative* – as opposed to *procedural* – programming. Ideally, a programmer should need only to be concerned with the declarative meaning of his program, while the procedural aspects of program's execution are handled automatically. The Prolog language became the privileged vehicle approximating this ideal. The first Prolog compiler [WPP77] showed that it could be a practical language and helped disseminate it worldwide.

Clearly, this ideal cannot possibly be fulfilled without a precise definition of proper declarative semantics of logic programs and, in particular, of the meaning of negation in logic programming. Logic programs, however, do not use classical logical negation, but rely instead on a non-monotonic operator, often referred to as "negation by failure" or "negation by default". The non-monotonicity of this operator allows us to view logic programs as special non-monotonic theories, and thus makes it possible to draw from the extensive research in the area of non-monotonic reasoning and use it as guidance in the search for a suitable semantics for logic programs.

On the other hand, while logic programs constitute only a subclass of the class of all non-monotonic theories, they are sufficiently expressive to allow formalizations of many important problems in non-monotonic reasoning and provide fertile testing grounds for new formalizations. Moreover, since logic programs admit relatively efficient computational mechanisms, they can be used as inference engines for non-monotonic formalisms.

The development of formal foundations of logic programming began in the late 1970s, especially with the works [EK76,Cla78,Rei78b]. Further progress in this direction was achieved in the early 1980's, leading to the appearance of the first book on the foundations of logic programming [Llo87]. The selection of logic programming as the underlying paradigm for the Japanese Fifth Generation Computer Systems Project led to the rapid proliferation of various logic programming languages.

In parallel, starting at about 1980, *non–monotonic Reasoning* entered into computer science and began to constitute a new field of active research. It was originally initiated because *Knowledge Representation* and *Common-Sense Reasoning* using classical logic came to its limits. Formalisms like classical logic are inherently monotonic and they seem to be too weak and therefore inadequate for such reasoning problems.

In recent years, independently of the research in logic programming, people interested in knowledge representation and non–monotonic reasoning also tried to define declarative semantics for programs containing *default* or *explicit* negation and even *disjunctions*. They defined various semantics by appealing to (different) intuitions they had about programs.

Due to logic programming's declarative nature, and amenability to implementation, it quickly became a prime candidate for knowledge representation. Its adequateness became more apparent after the relationships established in the mid 1980's between logic programs and deductive databases [Rei84,GMN84,LT85], [LT86,Min88].

The use of both logic programming and deductive databases for knowledge representation is based on the so called *"logical approach to knowledge representation"*. This approach rests on the idea of providing machines with a logical specification of the knowledge that they possess, thus making it independent of any particular implementation, context–free, and easy to manipulate and reason about.

Consequently, a precise meaning (or semantics) is associated with any logic program in order to provide its declarative specification. The performance of any computational mechanism is then evaluated by comparing its behaviour to the specification provided by the declarative semantics. Finding a suitable declarative semantics for logic programs has been acknowledged as one of the most important and difficult research areas of logic programming.

From the implementational point of view, for some time now programming in logic has been shown to be a viable proposition. As a result of the effort to find simple and efficient theorem proving strategies, Horn clause programming under SLD resolution was discovered and implemented [KK71,CKPR73].

However, because Horn clauses admit only positive conclusions or facts, they give rise to a monotonic semantics, i.e. one by which previous conclusions are never questioned in spite of additional information, and thus the number of derived conclusions cannot decrease – hence the monotonicity.

Horn clause programming augmented with the NOT operator (i.e. Prolog), under the SLDNF derivation procedure [Llo87], does allow negative conclusions; these, however, are only drawn by default (or implicitly), just in case the corresponding positive conclusion is not forthcoming in a finite number of steps, taking the program as it stands. This condition also prevents, by definition, the appearance of any and all contradictions, and so does not allow for reasoning about contradictory information.

The NOT form of negation is capable of dealing with incomplete information, by assuming false exactly what is not true in a finite manner – hence the Closed World Assumption (CWA) [Rei78a] or completion semantics given to such programs [Cla78]. However, irrespective of other problems with this semantics, there remains the issue of the proper treatment of non-terminating computations, even for finite programs.

To deal with the difficulties faced by the completion semantics, a spate of semantic proposals were set forth from the late eigthies onwards, of which the well–founded semantics of [GRS91] is an outcome. It deals semantically with non-terminating computations, by assigning them the truth value "undefined", thereby giving semantics to every program. Moreover, it enjoys a number of desirable structural properties [Dix92a,Dix92b,Dix91,Dix95a,Dix95b,BD96a].

However, the well–founded semantics deals solely with normal programs, i.e. those with only negation by default, and thus it provides no mechanism for explicitly declaring the falsity of literals. This can be a serious limitation. In fact, recently several authors have stressed and shown the importance of including in logic programs a symmetric form of negation, denoted $\neg$, with respect to default negation [APP96], for use in deductive databases, knowledge representation, and non–monotonic reasoning [GL91,GL92,Ino91,Kow90,KS90,PW90,PAA91b,AP96], [PAA92,PDA93,Wag91,PAA91c]. This symmetric negation may assume several garbs and names (**classical, pseudo, strong, explicit*) corresponding to different properties and distinct applicability. Default and symmetric negation can be related: indeed some authors uphold the *coherence principle*, which stipulates the latter entails the former.

So-called *extended* logic programs, those which introduce a symmetric form of negation, both in the body and conclusion of rules, have been instrumental in bridging logic programming and non–monotonic reasoning formalisms, such as *Default Logic*, *Auto–Epistemic Logic*, and *Circumscription*: non–monotonic reasoning formalisms provide elegant semantics for logic programming, in particular in what regards the meaning of negation as failure (or by default); non–monotonic reasoning formalisms help one understand how logic programming can be used to formalize and solve several problems in AI. On the other hand, non–monotonic reasoning formalisms benefit from the existing procedures of logic programming; and, finally, new problems of non–monotonic reasoning are raised and solved by logic programming.

Of course, introducing symmetric negation conclusions requires being able to deal with contradiction. Indeed, information is not only normally incomplete but contradictory to boot. Consequently, not all negation by default assumptions might be made, but only those not partaking of contradiction. This is tantamount to the ancient and venerable logical principle of "reductio ad absurdum": *if an assumption leads to contradiction withdraw it.* One major contribution of work on extended logic programming is that of tackling this issue.

Whereas default negation, and the revision of believed assumptions in the face of contradiction, are the two non–monotonic reasoning mechanisms available in logic programming, their use in combination with symmetric negation adds on a qualitative representational expressivity that can capture a wide variety of logical reasoning forms, and serve as an instrument for programming them. The application domains facilitated by extended logic programming include for example taxonomies with exceptions and preferences, reasoning about actions, model based diagnosis, belief revision, updates, and declarative debugging.

It should be noted that symmetric negation differs from classical negation. In particular, the principle of the excluded middle is not adopted, and so neither is case analysis, whereby if p if q, and if p if $\neg q$, then p. Indeed, propositions are not just true or false, exclusively. For one, they may be both true and false. Moreover, once contradiction is removed, even so a proposition and its negation may both remain undefined. In fact, truth in logic programming should be taken in an auto-epistemic sense: truth is provability from an agent's knowledge, and possibily neither a proposition nor its negation might be provable from its present knowledge – their truth-value status' might be undefined for both. Hence case analysis is not justified: p may rest undefined if q or $\neg q$ are undefined.

This is reasonable because the truth of q is not something that either holds or not, inasmuch as it can refer to the agent's ability to deduce q, or some other agent's view of q. For that matter, the supposition that either q holds or does not hold might be contradictory with the rest of the agent's knowledge in either case.

Also, the procedural nature of logic programming requires that each conclusion be supported by some identifiable rule with a true body whose conclusion it is, not simply by alternatively applicable rules, as in case analysis. Conclusions must be procedurally grounded on known facts. This requirement is conducive to a sceptical view of derived knowledge, which disallows jumping to conclusions when that is not called for.

Next we make a short technical overview of the main results in the last 15 years in the area of logic program's declarative semantics. This overview is divided into four sections. In the first we review some of the most important semantics of normal logic programs while in the second we motivate the need of extending logic programming with a second kind of negation, and overview recent semantics for such extended programs. The third and forth section are very compact and not as detailed as the previous ones. In the third section we treat disjunctive logic programming and in the fourth section we give an overview of implemented systems and where they can be obtained.

3 Normal logic programs

Several recent overviews of normal logic programming semantics can be found in the literature (e.g. [She88,She90,PP90,Mon92,AB94,Dix95c,BD96b]). Here, for the sake of this text's self–sufficiency and to introduce some motivation, we distill a brief overview of the subject. In some parts we follow closely the overview of [PP90].

The structure of this section is as follows: first we present the language of normal logic programs and give some definitions needed in the sequel. Then we briefly review the first approaches to the semantics of normal programs and point out their problems. Finally, we expound in greater detail two of the more recent and important proposals, namely stable models and well–founded semantics.

3.1 Language

By an alphabet $\mathcal{A}$ of a language $\mathcal{L}$ we mean a (finite or countably infinite) disjoint set of constants, predicate symbols, and function symbols. In addition, any alphabet is assumed to contain a countably infinite set of distinguished variable symbols. A *term* over $\mathcal{A}$ is defined recursively as either a variable, a constant or an expression of the form $f(t_1, \ldots, t_n)$, where f is a function symbol of $\mathcal{A}$, and the t_i are terms. An atom over $\mathcal{A}$ is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol of $\mathcal{A}$, and the t_is are terms. A literal is either an atom A or its negation *not* A. We dub default literals those of the form *not* A.

A term (resp. *atom*, *literal*) is called *ground* if it does not contain variables. The set of all ground terms (resp. atoms) of $\mathcal{A}$ is called the *Herbrand universe* (resp. base) of $\mathcal{A}$. For short we use $\mathcal{H}$ to denote the Herbrand base of $\mathcal{A}$.

A *normal* logic program is a finite set of rules of the form:

$$H \leftarrow L_1, \ldots, L_n \qquad (n \geq 0)$$

where H is an atom and each of the L_is is a literal. In conformity with the standard convention we write rules of the form $H \leftarrow$ also simply as H.

A normal logic program P is called *definite* if none of its rules contains default literals. We assume that the alphabet $\mathcal{A}$ used to write a program P consists precisely of all the constants, and predicate and function symbols that explicitly appear in P. By Herbrand universe (resp. base) of P we mean the Herbrand universe (resp. base) of $\mathcal{A}$.

By *grounded version* of a normal logic program P we mean the (possibly infinite) set of ground rules obtained from P by substituting in all possible ways each of the variables in P by elements of its Herbrand universe.

In this work we restrict ourselves to Herbrand interpretations and models[2]. Thus, without loss of generality (cf. [PP90]), we coalesce a normal logic program P with its grounded version.

3.2 Interpretations and models

Next we define 2 and 3–valued Herbrand interpretations and models of normal logic programs. Since non–Herbrand interpretations are beyond the scope of this work, in the sequel we sometimes drop the qualification Herbrand.

Definition 1 (Two–valued interpretation). A 2–valued interpretation I of a normal logic program P is any subset of the Herbrand base $\mathcal{H}$ of P.

Clearly, any 2–valued interpretation I can be equivalently viewed as a set

$$T \cup not\ F$$ [3]

where $T = I$ and is the set of atoms which are true in I, and $F = \mathcal{H} - T$ is the set of atoms which are false in I. These interpretations are called 2–valued because in them each atom is either true or false, i.e. $\mathcal{H} = T \cup F$.

As argued in [PP90], interpretations of a given program P can be thought of as "possible worlds" representing possible states of our knowledge about the meaning of P. Since that knowledge is likely to be incomplete, we need the ability to describe interpretations in which some atoms are neither true nor false but rather undefined, i.e. we need 3–valued interpretations:

Definition 2 (Three–valued interpretation). By a 3–valued interpretation I of a program P we mean a set

$$T \cup not\ F$$

[2] For the subject of semantics based on non–Herbrand models, and solutions to the problems resulting from always keeping Herbrand models see e.g. [Kun87,Prz89c,GRS91].

[3] Where *not* $\{a_1, \ldots, a_n\}$ stands for $\{not\ a_1, \ldots, not\ a_n\}$.

where T and F are disjoint subsets of the Herbrand base $\mathcal{H}$ of P.

The set T (the T-part of I) contains all ground atoms true in I, the set F (the F-part of I) contains all ground atoms false in I, and the truth value of the remaining atoms is unknown (or undefined).

It is clear that 2–valued interpretations are a special case of 3–valued ones, for which $\mathcal{H} = T \cup F$ is additionally imposed.

Proposition 3. *Any interpretation $I = T \cup not\ F$ can equivalently be viewed as a function $I : \mathcal{H} \rightarrow V$ where $V = \{0, \frac{1}{2}, 1\}$, defined by:*

$$I(A) = \begin{cases} 0 & if\ \ not\ A \in I \\ 1 & if\ \ A \in I \\ \frac{1}{2} & \quad otherwise \end{cases}$$

Of course, for 2–valued interpretations there is no atom A such that $I(A) = \frac{1}{2}$. Models are defined as usual, and based on a truth valuation function:

Definition 4 (Truth valuation). If I is an interpretation, the truth valuation $\hat{I}$ corresponding to I is a function $\hat{I} : C \rightarrow V$ where C is the set of all formulae of the language, recursively defined as follows:

- if A is a ground atom then $\hat{I}(A) = I(A)$.
- if S is a formula then $\hat{I}(not\ S) = 1 - \hat{I}(S)$.
- if S and V are formulae then
  - $\hat{I}((S, V)) = min(\hat{I}(S), \hat{I}(V))$.
  - $\hat{I}(V \leftarrow S) = 1$ if $\hat{I}(S) \leq \hat{I}(V)$, and 0 otherwise.

Definition 5 (Three–valued model). A 3–valued interpretation I is called a 3–valued model of a program P iff for every ground instance of a program rule $H \leftarrow B$ we have $\hat{I}(H \leftarrow B) = 1$.

The special case of 2–valued models has the following straightforward definition:

Definition 6 (Two–valued model). A 2–valued interpretation I is called a 2–valued model of a program P iff for every ground instance of a program rule $H \leftarrow B$ we have $\hat{I}(H \leftarrow B) = 1$.

Some orderings among interpretations and models will be useful:

Definition 7 (Classical ordering). If I and J are two interpretations then we say that $I \leq J$ if $I(A) \leq J(A)$ for any ground atom A. If $\mathcal{I}$ is a collection of interpretations, then an interpretation $I \in \mathcal{I}$ is called minimal in $\mathcal{I}$ if there is no interpretation $J \in \mathcal{I}$ such that $J \leq I$ and $I \neq J$. An interpretation I is called least in $\mathcal{I}$ if $I \leq J$ for any other interpretation $J \in \mathcal{I}$. A model M of a program P is called minimal (resp. least) if it is minimal (resp. least) among all models of P.

Definition 8 (Fitting ordering). If I and J are two interpretations then we say that $I \leq_F J$ [Fit85] iff $I \subseteq J$. If $\mathcal{I}$ is a collection of interpretations, then an interpretation $I \in \mathcal{I}$ is called F-minimal in $\mathcal{I}$ if there is no interpretation $J \in \mathcal{I}$ such that $J \leq_F I$ and $I \neq J$. An interpretation I is called F-least in $\mathcal{I}$ if $I \leq_F J$ for any interpretation $J \in \mathcal{I}$. A model M of a program P is called F-minimal (resp. F-least) if it is F-minimal (resp. F-least) among all models of P.

Note that the classical ordering is related with the amount of true atoms, whereas the Fitting ordering is related with the amount of information, i.e. nonundefinedness.

3.3 Clark's and Fitting's Semantics

As argued above, a precise meaning or semantics must be associated with any logic program, in order to provide a declarative specification of it. Declarative semantics provides a mathematically precise definition of the meaning of a program, which is independent of its procedural executions, and is easy to manipulate and reason about.

In contrast, procedural semantics is usually defined as a procedural mechanism that is capable of providing answers to queries. The correctness of such a mechanism is evaluated by comparing its behaviour to the specification provided by the declarative semantics. Without the latter, the user needs an intimate knowledge of the procedural aspects in order to write correct programs.

The first attempt to provide a declarative semantics to logic programs is due to [EK76], and the main motivation behind their approach is based on the idea that one should minimize positive information as much as possible, limiting it to facts explicitely implied by a program, making everything else false. In other words, their semantics is based on a natural form of *"closed world assumption"* [Rei78a].

Example 9. Consider program P :

$$\begin{array}{c} able\_mathematician(X) \leftarrow physicist(X) \\ physicist(einstein) \\ president(soares) \end{array}$$

This program has several (2–valued) models, the largest of which is the model where both Einstein and Soares are at the same time presidents, physicists and able mathematicians. This model does not correctly describe the intended meaning of P, since there is nothing in P to imply that Soares is a physicist or that Einstein is a president. In fact, the lack of such information should instead indicate that we can assume the contrary.

This knowledge is captured by the least (2–valued) model of P :

$$\{physicist(einstein), able\_mathematician(einstein), president(soares)\}$$

The existence of a unique least model for every definite program (proven in [EK76]), led to the definition of the so called *"least model semantics"* for definite programs. According to that semantics an atom A is true in a program P iff it belongs to the least model of P; otherwise A is false.

It turns out that this semantics does not apply to programs with default negation. For example, the program $P = \{p \leftarrow not\ q\}$ has two minimal models, namely $\{p\}$ and $\{q\}$. Thus no least model exists.

In order to define a declarative semantics for normal logic programs with negation as failure[4], [Cla78] introduced the so–called *"Clark's predicate completion"*. Informally, the basic idea of completion is that in common discourse we often tend to use "if" statements when we really mean "iff" ones. For instance, we may use the following program P to describe the natural numbers:

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

This program is too weak. It does not imply that nothing but $0, 1, \ldots$ is a natural number. In fact what we have in mind regarding program P is:

$$natural\_number(X) \Leftrightarrow (X = 0 \vee (\exists Y \mid X = succ(Y) \wedge natural\_number(Y)))$$

Based on this idea Clark defined the completion of a program P, the semantics of P being determined by the 2–valued models of its completion.

However Clark's completion semantics has some serious drawbacks. One of the most important is that the completion of consistent programs may be inconsistent, thus failing to assign to those programs a meaning. For example the completion of the program $\{p \leftarrow not\ p\}$ is $\{p \Leftrightarrow not\ p\}$, which is inconsistent.

In [Fit85], the author showed that the inconsistency problem for Clark's completion can be elegantly eliminated by considering 3–valued models instead of 2–valued ones. This led to the definition of the so–called *"Fitting semantics"* for normal logic programs. In [Kun87], Kunen showed that this semantics is not recursively enumerable, and proposed a modification.

Unfortunately, "Fitting's semantics" inherits several problems of Clark's completion, and in many cases leads to a semantics that appears to be too weak. This issue has been extensively discussed in the literature (see [She88,Prz89c,GRS91], [Dix95c,BD96b]). Forthwith we illustrate some of these problems with the help of examples:

Example 10. Consider[5] program P :

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

[4] Here we adopt the designation of *"negation by default"*. Recently, this designation has been used in the literature instead of the more operational *"negation as failure"*.

[5] This example first appeared in [GRS91].

that describes which vertices are reachable from a given vertice a in a graph.

Fitting semantics cannot conclude that vertices c and d are not reachable from a. Here the difficulty is caused by the existence of the symmetric rules $edge(c, d)$, and $edge(d, c)$.

Example 11. Consider P :

$$\begin{aligned} &bird(tweety) \\ fly(X) &\leftarrow bird(X), not\ abnormal(X) \\ abnormal(X) &\leftarrow irregular(X) \\ irregular(X) &\leftarrow abnormal(X) \end{aligned}$$

where the last two rules just state that "irregular" and "abnormal" are synonymous.

Based on the fact that nothing leads us to the conclusion that tweety is abnormal, we would expect the program to derive *not abnormal(tweety)*, and consequently that it flies. But Clark's completion of P is:

$$\begin{aligned} bird(X) &\Leftrightarrow X = tweety \\ fly(X) &\Leftrightarrow bird(X), not\ abnormal(X) \\ abnormal(X) &\Leftrightarrow irregular(X) \end{aligned}$$

from which it does not follow that tweety isn't abnormal.

It is worth noting that without the last two rules both Clark's and Fitting's semantics yield the expected result.

One possible explanation for such a behaviour is that the last two rules lead to a loop. This explanation is procedural in nature. But it was the idea of replacing procedural programming by declarative programming that brought about the concepts of logic programming in first place and so, as argued in [PP90], it seems that such a procedural explanation should be rejected.

The problems mentioned above are caused by the difficulty in representing transitive closure using completion. In [Kun90] it is formally showed that both Clark's and Fitting's semantics are not sufficiently expressive to represent transitive closure.

In order to solve these problems some model–theoretic approaches to declarative semantics have been defined. In the beginning, such approaches did not attempt to give a meaning to every normal logic program. On the contrary, they were based on syntactic restrictions over programs, and only program complying with such restrictions were given a semantics. Examples of syntactically restricted program classes are stratified [ABW88], locally stratified [Prz89c] and acyclic [AB91], and examples of semantics for restricted programs are the perfect model semantics [ABW88,Prz89c,Gel89b], and the weakly perfect model semantics [PP88]. Here we will not review any of these approaches. For their overview, the reader is referred to e.g. [PP90].

3.4 Stable model semantics

In [GL88], the authors introduce the so–called *"stable model semantics"*. This model–theoretic declarative semantics for normal programs generalizes the previously referred semantics for restricted classes of programs, in the sense that for such classes the results are the same and, moreover, for some non–restricted programs a meaning is still assigned.

The basic ideas behind the stable model semantics came from the field of non–monotonic reasoning formalism. There, literals of the form $not\ A$ are viewed as default literals that may or may not be assumed or, alternatively, as epistemic literals $\sim\mathcal{B}A$ expressing that A is not believed.

Informally, when one assumes true some set of (hypothetical) default literals, and false all the others, some consequences follow according to the semantics of definite programs [EK76]. If the consequences completely corroborate the hypotheses made, then they form a stable model. Formally:

Definition 12 (Gelfond–Lifschitz operator). Let P be a normal logic program and I a 2–valued interpretation. The GL–transformation of P modulo I is the program $\frac{P}{I}$ obtained from P by performing the following operations:

- remove from P all rules which contain a default literal $not\ A$ such that $A \in I$;
- remove from the remaining rules all default literals.

Since $\frac{P}{I}$ is a definite program, it has a unique least model J. We define $\Gamma(I) = J$.

It turns out that fixed points of the Gelfond–Lifschitz operator Γ for a program P are always models of P. This result led to the definition of stable model semantics:

Definition 13 (Stable model semantics). A 2–valued interpretation I of a logic program P is a stable model of P iff $\Gamma(I) = I$.

An atom A of P is true under the stable model semantics iff A belong to all stable models of P.

One of the main advantages of stable model semantics is its close relationship with known non–monotonic reasoning formalisms:

As proven in [BF88], the stable models of a program P are equivalent to Reiter's default extensions [Rei80] of the default theory obtained from P by identifying each program rule:

$$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

with the default rule:

$$\frac{B_1, \ldots, B_n\ : \sim C_1, \ldots, \sim C_m}{H}$$

where $\sim$ denotes classical negation.

Moreover, from the results of [Gel87], it follows directly that stable models are equivalent to Moore's autoepistemic expansions [Moo85] of the theory obtained by replacing in P every default literal *not* A by $\sim\mathcal{B}A$ and then reinterpreting the rule connective $\leftarrow$ as material implication.

In spite of the strong relationship between logic programming and non–monotonic reasoning, in the past these research areas were developing largely independently of one another, and the exact nature of their relationship was not closely investigated or understood.

The situation has changed significantly with the introduction of stable models, and the establishment of formal relationships between these and other non–monotonic formalisms. In fact, in recent years increasing and productive effort has been devoted to the study of the relationships between logic programming and several non–monotonic reasoning formalisms. As a result, international workshops have been organized, in whose proceedings many works and references to the theme can be found [NMS91,PN93,MNT95,DFN97].

Such relationships turn out to be mutual beneficial. On the one hand, non–monotonic formalisms provide elegant semantics for logic programming, specially in what regards the meaning of default negation (or negation as failure), and help one understand how logic programs can be used to formalize several types of reasoning in Artificial Intelligence. On the other hand, those formalisms benefit from the existing procedures of logic programming, and some new issues of the former are raised and solved by the latter. Moreover, relations among non–monotonic formalisms themselves have been facilitated and established via logic programming.

3.5 Well–founded semantics

Despite its advantages, and of being defined for more programs than any of its predecessors, stable model semantics still has some important drawbacks:

- First, some programs have no stable models. One such program is $P = \{a \leftarrow not\ a\}$.
- Even for programs with stable models, their semantics do not always lead to the expected intended results. For example consider program P :

$$\begin{aligned} a &\leftarrow not\ b \\ b &\leftarrow not\ a \\ c &\leftarrow not\ a \\ c &\leftarrow not\ c \end{aligned}$$

  whose only stable model is $\{c, b\}$. Thus b and c are consequences of the stable model semantics of P. However, if one adds c to P as a lemma, the semantics of P changes, and b no longer follows. This issue is related with the property of *cumulativity* ([Dix95a]).
- Moreover, it is easy to see that it is impossible to derive b from P using any derivation procedure based on top–down (SL–like) rewriting techniques.

This is because such a procedure, beginning with the goal $\leftarrow b$ would reach only the first two rules of P, from which b cannot be derived. This issue is related with the property of *relevance* ([Dix95b]).
- The computation of stable models is NP–complete [MT91] even within simple classes of programs, such as propositional logic programs. This is an important drawback, specially if one is interested in a program for efficiently implementing knowledge representation and reasoning.
- Last but not least, by always insisting on 2–valued interpretations, stable model semantics often lack expressivity.

The well–founded semantics was introduced in [GRS91], and overcomes all of the above problems. This semantics is also closely related to some of the major non–monotonic formalisms ([AP96]).

Many different equivalent definitions of the well–founded semantics exist (e.g. [Prz89a,Prz89b,Bry89,PP90,Dun91,Prz91a,BS91,Mon92,Dix95b,BD97b]). Here we use the definition introduced in [PP90] because, in our view, it is the one more technically related with the definition of stable models above[6]. Indeed, it consists of a natural generalization for 3–valued interpretations of the stable model semantics. In its definition the authors begin by introducing 3–valued (or partial) stable models, and then show that the F–least of those models coincides with the well–founded model as first defined in [GRS91].

In order to formalize the notion of partial stable models, Przymusinski first expand the language of programs with the additional propositional constant $\mathbf{u}$ with the property of being undefined in every interpretation. Thus they assume that every interpretation I satisfies:

$$\hat{I}(\mathbf{u}) = \hat{I}(not\ \mathbf{u}) = \frac{1}{2}$$

A non–negative program is a program whose premises are either atoms or $\mathbf{u}$. In [PP90], it is proven that every non–negative program has a 3–valued least model. This led to the following generalization of the Gelfond–Lifschitz Γ–operator:

Definition 14 (Γ^*–operator). Let P be a normal logic program, and let I be a 3–valued interpretation. The extended GL–transformation of P modulo I is the program $\frac{P}{I}$ obtained from P by performing the operations:

- remove from P all rules which contain a default literal *not* A such that $I(A) = 1$;
- replace in the remaining rules of P those default literals *not* A such that $I(A) = \frac{1}{2}$ by $\mathbf{u}$;
- remove from the remaning rules all default literals.

Since the resulting program is non–negative, it has a unique 3–valued least model J. We define $\Gamma^*(I) = J$.

[6] For a more practical introduction to the well–founded semantics the reader is referred to [PAA91a]. A detailed exposition based on abstract properties is given in [BD96b].

Definition 15 (Well–founded semantics). A 3–valued interpretation I of a logic program P is a partial stable model of P iff $\Gamma^*(I) = I$.

The well–founded semantics of P is determined by the unique F–least partial stable model of P, and can be obtained by the (bottom–up) iteration of Γ^* starting from the empty interpretation.

An alternative equivalent definition, by means of the squared Γ operator, is given as a special case in the section on well–founded semantics with explicit negation below.

4 Extended logic programs

Recently several authors have stressed and shown the importance of including, beside default negation, a symmetric second kind of negation $\neg$ in logic programs [APP96], for use in deductive databases, knowledge representation, and non–monotonic reasoning [GL91,GL92,Ino91,Kow90,KS90,PW90,PAA91b,PAA91c], [PAA92,PDA93,Wag91]. This symmetric form of negation may assume several garbs and names (**classical, pseudo, strong, explicit*) corresponding to different properties and distinct applicability. Default and symmetric negation can be related: indeed some authors uphold the *coherence principle*, which stipulates that the latter entails the former.

In this section we begin by reviewing the main motivations for introducing this kind of symmetric negation in logic programs. Then we define an extension of the language of programs to two negations, and briefly overview the main proposed semantics for these programs.

4.1 Motivation

In normal logic programs the negative information is implicit, i.e. it is not possible to explicitly state falsity, and propositions are assumed false if there is no reason to believe they are true. This is what is wanted in some cases. For instance, in the classical example of a database that explicitly states flight connections, one wants to implicitly assume that the absence of a connection in the database means that no such connection exists.

However this is a serious limitation in other cases. As argued in [PW90,Wag91], explicit negative information plays an important rôle in natural discourse and commonsense reasoning. The representation of some problems in logic programming would be more natural if logic programs had some way of explicitly representing falsity. Consider for example the statement:

"Penguins do not fly"

One way of representing this statement within logic programming could be:

$$no\_fly(X) \leftarrow penguin(X)$$

or equivalently:

$$fly'(X) \leftarrow penguin(X)$$

as suggested in [GL89].

But these representations do not capture the connection between the predicate $no\_fly(X)$ and the predication of flying. This becomes clearer if, additionally, we want to represent the statement:

"Birds fly"

Clearly this statement can be represented by

$$fly(X) \leftarrow bird(X)$$

But then, no connection whatsoever exists between the predicates $no\_fly(X)$ and $fly(X)$. Intuitively one would like to have such an obvious connection established.

The importance of these connections grows if we think of negative information for representing exceptions to rules [Kow90]. The first statement above can be seen as an exception to the general rule that normally birds fly. In this case we really want to establish the connection between flying and not flying.

Exceptions expressed by sentences with negative conclusions are also common in legislation [Kow89,Kow92]. For example, consider the provisions for depriving British citizens of their citizenship:

> *40 - (1) Subject to the provisions of this section, the Secretary of State may by order deprive any British citizen to whom this subsection applies of his British citizenship if [...].*
>
> *(5) The Secretary of State shall not deprive a person of British citizenship under this section if [...].*

Clearly, *40 (1)* has the logical form "P if Q" whereas *40 (5)* has the form "¬ P if R". Moreover, it is also clear that *40 (5)* is an exception to the rule of *40 (1)*

Above we argued for the need of having a symmetric negation in the head of rules. But there are also reasons that compel us to believe symmetric negation is needed also in their bodies. Consider the statement[7]:

" A school bus may cross railway tracks under the condition that there is no approaching train"

It would be wrong to express this statement by the rule:

$$cross \leftarrow not\ train$$

The problem is that this rule allows the bus to cross the tracks when there is no information about either the presence or the absence of a train. The situation is different if symmetric negation is used:

$$cross \leftarrow \neg train$$

[7] This example is due to John McCarthy, and was published for the first time in [GL91].

Then the bus is only allowed to cross the tracks if the bus driver is sure that there is no approaching train. The difference between *not* p and $\neg p$ in a logic program is essential whenever we cannot assume that available positive information about p is complete, i.e. we cannot assume that the absence of information about p clearly denotes its falsity.

Moreover, the introduction of symmetric negation in combination with the existing default negation allows for greater expressivity, and so for representing statements like:

" If the driver is not sure that a train is not approaching then he should wait"

in a natural way:

$$wait \leftarrow not\ \neg train$$

Examples of such combinations also appear in legislation. For example consider the following article from "The British Nationality Act 1981" [HMS81]:

> *(2) A new–born infant who, after commencement, is found abandoned in the United Kingdom shall acquire british citizenship by section 1.2 if it is not shown that it is not the case that the person is born [...]*

Clearly, conditions of the form "it is not shown that it is not the case that P" can be expressed naturally by *not* $\neg P$.

Another motivation for introducing symmetric negation in logic programs relates to a desired symmetry between positive and negative information. This is of special importance when the negative information is easier to represent than the positive one. One can first represent it negatively, and then say that the positive information corresponds to its complement.

In order to make this clearer, take the following example [GL91]:

Example 16. Consider a graph description based on the predicate $arc(X, Y)$, which expresses that in the graph there is an arc from vertice X to vertice Y. Now suppose that we want to determine which vertices are terminals. Clearly, this is a case where the complement information is easier to represent, i.e. it is much easier to determine which vertices are not terminal. By using symmetric negation in combination with negation by default, one can then easily say that terminal vertices are those which are not nonterminal:

$$\begin{aligned} \neg terminal(X) &\leftarrow arc(X, Y) \\ terminal(X) &\leftarrow not\ \neg terminal(X) \end{aligned}$$

Finally, another important motivation for extending logic programming with symmetric negation is to generalize the relationships between logic programs and non–monotonic reasoning formalisms.

As mentioned in section 3.3, such relationships, drawn for the most recent semantics of normal logic programs, have proven of extreme importance for both sides, giving them mutual benefits and clarifications. However, normal logic programs just map into narrow classes of the more general non–monotonic formalisms. For example, simple default rules such as (where "$\sim$" is classical negation):

$$\frac{\sim a \ : \sim b}{c} \qquad\qquad \frac{a \ : \ b}{c} \qquad\qquad \frac{a \ : \ b}{\sim c}$$

cannot be represented by a normal logic program. Note that not even normal nor semi–normal default rules can be represented using normal logic programs. This is so because these programs cannot represent rules with negative conclusions, and normal rules with positive conclusions have also positive justifications, which is impossible in normal programs.

Since, as shown below, extended logic programs also bear a close relationship to non–monotonic reasoning formalisms, they improve on those of normal programs as extended programs map into broader classes of theories in non–monotonic formalisms, and so more general relations between several of those formalisms can now be made via logic programs.

One example of such an improvement is that the introduction of symmetric negation into logic programs makes it possible to represent normal and semi–normal defaults within logic programming. On the one side, this provides methods for computing consequences of normal default theories. On the other, it allows for the appropriation in logic programming of work done using such theories for representing knowledge.

4.2 Language of extended programs

As for normal logic programs, an atom over an alphabet $\mathcal{A}$ is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol, and the t_is are terms. In order to extend our language with a second kind of negation, we additionally define an objective literal over $\mathcal{A}$ as being an atom A or its symmetric negation $\neg A$. We also use the symbol $\neg$ to denote complementary literals in the sense of symmetric negation. Thus $\neg\neg A = A$. Here, a literal is either an objective literal L or its default negation *not* L. We dub default literals those of the form *not* L.

By the extended Herbrand base of $\mathcal{A}$, we mean the set of all ground objective literals of $\mathcal{A}$. Whenever unambigous we refer to the extended Herbrand base of an alphabet, simply as Herbrand base, and denote it by $\mathcal{H}$.

An extended logic program is a finite set of rules of the form:

$$H \leftarrow L_1, \ldots, L_n \qquad (n \geq 0)$$

where H is an objective literal and each of the L_is is a literal. As for normal programs, if $n = 0$ we omit the arrow symbol.

By the extended Herbrand base $\mathcal{H}$ of P we mean the extended Herbrand base of the alphabet consisting of all the constants, predicate and function symbols that explicitly appear in P.

Interpretation is defined as for normal programs, but using the extended Herbrand base instead.

Whenever unambigous, we refer to extended logic programs simply as logic programs or programs. As in normal programs, a set of rules stands for all its ground instances.

4.3 Answer–sets semantics

The first semantics defined for extended logic programs was the so–called *"answer–sets semantics"* [GL91]. There the authors defined for the first time the language of logic programs with two kinds of negation – default negation *not* and what they called "classical" negation $\neg$, which is symmetric with respect to *not* .

The answer–sets semantics is a generalization of the stable model semantics for the language of extended programs. Roughly, an answer–set of an extended program P is a stable model of the normal program obtained from P by replacing objective literals of the form $\neg L$ by new atoms, say $\neg\_L$.

Definition 17 (The Γ–operator). Let P be an extended logic program and I a 2–valued interpretation. The GL–transformation of P modulo I is the program $\frac{P}{I}$ obtained from P by:

- first denoting every objective literal in $\mathcal{H}$ of the form $\neg A$ by a new atom, say $\neg\_A$;
- replacing in both P and I, these objective literals by their new denotation;
- then performing the following operations:
  - removing from P all rules which contain a default literal *not* A such that $A \in I$;
  - removing from the remaning rules all default literals.

Since $\frac{P}{I}$ is a definite program it has a unique least model J.

If J contains a pair of complementary atoms, say A and $\neg\_A$, then $\Gamma(I) = \mathcal{H}$.

Otherwise, let J' be the interpretation obtained from J by replacing the newly introduced atoms $\neg\_A$ by $\neg A$. We define $\Gamma(I) = J'$.

Definition 18 (Answer–sets semantics). A 2–valued interpretation I of an extended logic program P is an answer–set model of P iff $\Gamma(I) = I$.

An objective literal L of P is true under the answer–sets semantics iff L belongs to all answer–sets of P; L is false iff $\neg L$ is true; otherwise L is unknown.

In [GL91], the authors showed that the answer–sets of an extended program P are equivalent to Reiter's default extensions of the default theory obtained from P by identifying each program rule:

$$H \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m, not\ D_1, \ldots, not\ D_k, not\ \neg E_1, \ldots, not\ \neg E_j$$

with the default rule:

$$\frac{B_1, \ldots, B_n, \sim C_1, \ldots, \sim C_m\ : \sim D_1, \ldots, \sim D_k, E_1, \ldots, E_j}{H'}$$

where $H' = H$ if H is an atom, or $H' = \sim L$ if $H = \neg L$.

4.4 e–answer–sets semantics

Another semantics generalizing stable models for the class of extended programs is the e–answer–sets semantics of [KS90]. There, the authors claim that "classically" negated atoms in extended programs play the rôle of exceptions. Thus they impose a preference of negative over positive objective literals.

The e–answer–sets semantics is obtainable from the answer–sets semantics after a suitable program transformation. For the sake of simplicity, here we do not give the formal definition of e–answer–sets, but instead show its behaviour on an example:

Example 19. Consider program P :

$$\begin{aligned} fly(X) &\leftarrow bird(X) \\ \neg fly(X) &\leftarrow penguin(X) \\ bird(X) &\leftarrow penguin(X) \\ penguin(tweety) & \end{aligned}$$

This program allows for both the conclusions $fly(tweety)$ and $\neg fly(tweety)$. Thus its only answer–sets is $\mathcal{H}$.

In e–answer–sets semantics, since conclusions of the form $\neg L$ are preferred over those of the form L, $\neg fly(tweety)$ overrides the conclusion $fly(tweety)$, and thus

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

is an e–answer–set of P.

The rationale for this overriding is that the second rule is an exception to the first one.

4.5 Well–founded semantics with pseudo negation

In [Prz90], the author argues that the technique used in answer–sets for generalizing stable models is quite general. Based on that he defines a semantics which generalizes the well–founded semantics for the class of extended programs[8], as follows:

Definition 20 (Well–founded semantics with pseudo negation). A 3–valued interpretation I is a partial stable model of an extended logic program P iff I' is a partial stable model of the normal program P', where I' and P' are obtained respectively from I and P, by replacing every objective literal of the form $\neg A$ by a new atom, say $\neg\_A$.

The well–founded semantics with pseudo negation of P is determined by the unique F–least partial stable model of P.

[8] In the sequel we refer to this semantics as *"well–founded semantics with pseudo negation"*.

4.6 Nelson's strong negation semantics

Based on the notions of vivid logic [Lev86] and strong negation [Nel49], [Wag91] presents an alternative definition of the answer–sets semantics. There, the author claims that the $\neg$–negation of extended logic programs is not classical negation but rather Nelson's strong negation.

In fact, consider the following program P :

$$b \leftarrow a$$
$$b \leftarrow \neg a$$

If *real* classical negation were used then b would be a consequence of P, because for classical negation $a \vee \neg a$ is a tautology. However, in neither of the above mentioned semantics b follows from P.

4.7 Well–founded semantics with explicit negation

Many semantics for extended logic programs view default negation and symmetric negation as unrelated. To overcome this situation a new semantics for extended logic programs was proposed in [PA92,AP96]. Well-founded Semantics with Explicit Negation (WFSX for short) embeds a "coherence principle" providing the natural missing link between both negations: if $\neg L$ holds then $not\ L$ should too (similarly, if L then $not\ \neg L$). More recently, a paraconsistent extension of this semantics ($WFSX_p$) has been proposed in [ADP95,Dam96] via an alternating fixpoint definition that we now recapitulate.

As usual, for the sake of simplicity and without loss of generality, we will restrict the discussion to (possibly infinite) propositional, or to ground, programs. A non-ground program stands for its fully instantiated version, i.e. the set of all ground instances of its rules. The alphabet of the language of the programs is the set of atoms At. Atoms can be negated by juxtaposing them with the explicit negation symbol "$\neg$" thereby obtaining the explicitly negated literals. The explicit complement of a set $A = \{a_1, a_2, \ldots\}$ is $\neg A = \{\neg a_1, \neg a_2, \ldots\}$. The set of objective literals is $OLit = At \cup \neg At$. Default literals are of the form $not\ a$ and $not\ \neg a$, where a is an atomic proposition in the language's alphabet. The set of all literals is $Lit = OLit \cup not\ OLit$, where default negation of a set of objective literals stands for the set comprised of the default negation of each one.

Definition 21. An extended logic program is a set of rules of the form

$$L_0 \leftarrow L_1, \ldots, L_m, not\ M_1, \ldots, not\ M_n \qquad (m, n \geq 0)$$

where L_i $(0 \leq i \leq m)$ and M_j $(1 \leq j \leq n)$ are objective literals.

We begin by recalling the definition of Gelfond-Lifschitz Γ operator, used in the alternating fixpoint definition of WFS [Gel89a], $WFSX$, and $WFSX_p$.

To impose the coherence requirement Alferes and Pereira resort to the semi-normal version of an extended logic program.

Definition 22. [PA92,AP96] The semi-normal version P_s of a program P is obtained from P by adding to the (possibly empty) *Body* of each rule $L \leftarrow Body$ the default literal $not\ \neg L$, where $\neg L$ is the complement of L with respect to explicit negation.

The semi-normal version of a program introduces a new anti-monotonic operator: $\Gamma_{P_s}(S)$. Below we use $\Gamma(S)$ to denote $\Gamma_P(S)$, and $\Gamma_s(S)$ to denote $\Gamma_{P_s}(S)$.

Theorem 23. [ADP95] *The operator $\Gamma\Gamma_s$ is monotonic, for arbitrary sets of literals.*

Consequently every program has a least fixpoint of $\Gamma\Gamma_s$. This defines the semantics for paraconsistent logic programs. It also ensures that the semantics is well-defined, i.e. assigns meaning to every extended logic program.

Definition 24. [ADP95] Let P be an extended logic program and T a fixpoint of $\Gamma\Gamma_s$, then $T \cup not\ (\mathcal{H}_P - \Gamma_s T)$ is a paraconsistent partial stable model of P (PSM_p). The paraconsistent well-founded model of P, $WFM_p(P)$, is the least PSM_p under set inclusion order.

To enforce consistency on the paraconsistent partial stable models Alferes and Pereira need only insist the extra condition $T \subseteq \Gamma_s T$ be verified, which succintly guarantees that for no objective literal, L and $not\ L$ simultaneously hold. So it automatically rejects contradictory models where L and $\neg L$ are both true because, by coherence, they would also entail $not\ L$ and $not\ \neg L$. Therefore *WFSX*$_p$ generalizes *WFSX*, as it does not impose this additional condition. Furthermore, for normal logic programs, it coincides with *WFS*, which is definable as the least fixpoint of $\Gamma\Gamma$.

Example 25. Let P be the extended logic program:

$$a \leftarrow not\ b. \qquad b \leftarrow not\ c. \qquad b \leftarrow not\ \neg c. \qquad c. \qquad \neg c. \qquad d \leftarrow not\ d.$$

The sequence for determining the least fixpoint of $\Gamma\Gamma_s$ of program P is:

$$\begin{aligned} I_0 &= \{\} \\ I_1 &= \Gamma\Gamma_s\{\} && = \Gamma\{a,b,c,\neg c,d\} = \{c,\neg c\} \\ I_2 &= \Gamma\Gamma_s\{c,\neg c\} && = \Gamma\{a,d\} \qquad\quad\ = \{a,b,c,\neg c\} \\ I_3 &= \Gamma\Gamma_s\{a,b,c,\neg c\} && = \Gamma\{d\} \qquad\qquad = I_2 \end{aligned}$$

Thus $WFM_p(P) = \{a,b,c,\neg c, not\ a, not\ \neg a, not\ b, not\ \neg b, not\ c, not\ \neg c, not\ \neg d\}$.

One of the distinguishing features of *WFSX*$_p$ is that it does not enforce default consistency, i.e. a and $not\ a$ can be simultaneously true, in contradistinction to all other semantics. In the above example this is the case for literals a, b, c and $\neg c$. It is due to the adoption of the coherence principle: for instance, because a and $\neg a$ hold then, by coherence, $not\ \neg a$ and $not\ a$ must hold too.

5 Disjunctive Logic Programs

Recently, considerable interest and research effort[9] has been given to the problem of finding a suitable extension of the *logic programming paradigm* beyond the class of normal logic programs. In particular, considerable work has been devoted to the problem of defining natural extensions of logic programming that ensure a proper treatment of *disjunctive information*. However, the problem of finding a suitable semantics for disjunctive programs and databases proved to be far more complex than it is in the case of normal, non-disjunctive programs[10].

There are good reasons justifying this extensive research effort. In natural discourse as well as in various programming applications we often use *disjunctive statements.* One particular example of such a situation is *reasoning by cases.* Other obvious examples include:

- *Null values:* for instance, an age "around 30" can be 28, 29, 30, 31, or 32;
- *Legal rules:* the judge always has some freedom for his decision, otherwise he/she would not be needed; so laws cannot have unique models;
- *Diagnosis:* only at the end of a fault diagnosis do we know exactly which part of some machine was faulty but while we are searching, different possibilities exist;
- *Biological inheritance:* if the parents have blood groups A and 0, the child must also have one of these two blood groups (example from [Lip79]);
- *Natural language understanding:* here there are many possibilities for ambiguity and they are represented most naturally by multiple intended models;
- *Conflicts in multiple inheritance:* if we want to keep as much information as possible, we should assume disjunction of the inherited values [BL93].

As [EGM93,EG93,EGM94] have shown, formalisms promoting disjunctive reasoning are more *expressive* and they are also more *natural to use* since they permit direct translation of disjunctive statements from natural language and from informal specifications. The additional expressive power of disjunctive logic programs significantly simplifies the problem of *translation* of non-monotonic formalisms into logic programs, and, consequently, facilitates using logic programming as an *inference engine* for non-monotonic reasoning. Moreover, extensive recent work devoted to theoretic and algorithmic foundations of disjunctive programming, suggests that there are good prospects for *extending the logic programming paradigm* to disjunctive programs.

What then should be viewed as an "extension of logic programming"? We believe that in order to demonstrate that a class of programs can be justifiably called an *extension of logic programs* one should be able to argue that:

---

[9] It suffices just to mention several recent workshops on *Extensions of Logic Programming* specifically devoted to this subject ([DPP95,DHSH96,Wol94,DLMW96]).

[10] The book by Minker et. al. [LMR92] provides a detailed and well-organized account of the extensive research effort in this area. See also [Dix95c,Min93].

– the proposed *syntax* of such programs resembles the syntax of logic programs but it applies to a significantly broader class of programs, which includes the class of disjunctive logic programs as well as the class of logic programs with "classical" (or symmetric) negation;
– the proposed *semantics* of such programs constitutes an intuitively natural extension of the semantics of normal logic programs, which, when restricted to normal logic programs, coincides with one of the well-established semantics of normal logic programs;
– there exists a reasonably simple procedural mechanism allowing, at least in principle, to compute the semantics[11].

It would be also desirable for the proposed class of programs and their semantics to be a special case of a more general non-monotonic formalism which would clearly link it to other well-established non-monotonic formalisms.

Several approaches to the semantics of disjunctive logic programs have been recently proposed and studied in the literature, e.g. [LMR92,Ros92,RT88,GL91], [Dix92b], [BD94,BD97b,BD97a,BD96a,EG93,EGM93,BLM90,Prz91b,Prz95b], and [Prz95a,BDNP97,BDP96]. Since a more thorough discussion of disjunctive programming is beyond the scope of this brief introduction, we refer the reader to those papers, as well as to papers published in this volume, for more details.

6 Implementations

In this section we give a rough overview of what semantics have been implemented so far and where they are available.

It should be noted, that the first-order versions of all semantics investigated here are undecidable. Nevertheless it is an important task to have running systems that *can handle programs with free variables*, and *are Goal-Oriented*. To ensure *completeness* (or *termination*) we need then additional requirements like *allowedness* (to prevent floundering), and no function symbols.

Although these restrictions ensure the Herbrand-universe to be finite (and thus we are really considering a propositional theory) we think that such a system has great advantages over a system that can just handle ground programs. For a language $\mathcal{L}$, the fully instantiated program can be quite large and difficult to handle effectively.

The goal-orientedness (or *Relevance* as mentioned above) is also important — after all this was one reason for the success of SLD-Resolution. As noted above, such a goal-oriented approach is not possible for the stable semantics.

There are various commercial PROLOG-systems that perform variants of SLDNF-Resolution. But in what follows, we concentrate in the following on extensions of semantics introduced in this paper.

[11] Observe that while such a mechanism cannot even in principle be efficient, due to the inherent NP-completeness of the problem of computing answers to just positive disjunctive programs, it can be efficient when restricted to specific subclasses of programs and queries and it can allow efficient approximation methods for broader classes of programs.

Currently, a library of implemented logic programming systems and interesting test-cases for such systems is collected as a project of the artificial intelligence group at Koblenz. We refer to `<http://www.uni-koblenz.de/ag-ki/LP/>`.

6.1 Non-Disjunctive Semantics

There are many theoretical papers that deal with the problem of implementation ([BD93,KSS91,DN95,FLMS93]) but only a few running systems. The problem of handling and representing ground programs given a non-ground one has also been addressed [KNS94,KNS95,EGLS96].

In [BNNS93,BNNS94] the authors showed how the problem of computing stable models can be transformed to an Integer-Linear Programming Problem. This has been extended in [DM93] to disjunctive programs.

Inoue et. al. show in [IKH92] how to compute stable models by transforming programs into propositional theories and then using a model-generation theorem prover.

In Bern, Switzerland, a group around G. Jäger is building a non-monotonic reasoning system which incorporates various monotonic and non-monotonic logics. We refer to `http://lwbwww.unibe.ch:8080/LWBinfo.html`.

Extended logic programs under the well-founded semantics are considered by Pereira and his colleagues: [PAA93,ADP95,AP96]. The REVISE system, which deals with contradiction removal pro paraconsistent programs in this semantics, can be found in `<http://www.uni-koblenz.de/ag-ki/LP/>` too.

[NS96] describes an implementation of WFS and STABLE with a special eye on complexity.

The most advanced system has been implemented by David Warren and his group in Stony Brook based on OLDT-algorithm of [TS86]. They first developed a meta-interpreter (SLG, see [CW96]) in PROLOG and then directly modified the WAM for a direct implementation of WFS (XSB). They use tabling-methods and a mixture of top-Down and bottom-up evaluation to detect loops. Their system is complete and terminating for non-floundering DATALOG. It also works for general programs but termination is not guaranteed. This system is described in [CW93,CSW95,CW95], and is available by anonymous ftp from `ftp.cs.sunysb.edu/pub/XSB/`.

6.2 Disjunctive Semantics

There are theoretical descriptions of implementations that have not yet been implemented: [FM95,MR95,CL95].

Here are some implemented systems. Inoue et. al. show in [IKH92] how to compute stable models for extended disjunctive programs in a bottom-up-fashion using a theorem prover. The approach of Bell et. al. ([NNS91]) was used by Dix/Müller [DM93,DM92,Mül92] to implement versions of the stationary semantics of Przymusinski ([Prz91c].

Brass/Dix have implemented both D-WFS and DSTABLE for allowed DATALOG programs ([BD95][12]). An implementation of static semantics ([Prz95b]) is described in [BDP96][13].

Seipel has implemented in his DisLog-system various (modified versions of) semantics of Minker and his group. His system is publicly available at the URL `http://sunwww.informatik.uni-tuebingen.de:8080/dislog/dislog.tar.Z`.

Finally, there is the DisLoP project undertaken by the Artificial Intelligence Research Group at the University of Koblenz and headed by J. Dix and U. Furbach ([DF96,ADN97]). This project aims at extending certain theorem proving concepts, such as restart model elimination [BF94] and hyper tableaux [BFN96] calculi, for disjunctive logic programming. Information on the DisLoP project and related publications can be obtained from the WWW page `<http://www.uni-koblenz.de/ag-ki/DLP/>`.

Acknowledgements

We wish to thank our colleagues José Júlio Alferes, Stefan Brass, Gerhard Brewka, Carlos Damásio, and Halina Przymusinska for our use of material from joint publications. Luís Moniz Pereira acknowledges also the support of JNICT/PRAXIS and its funding of the project MENTAL (#2/2.1/TIT/1593/95).

References

[AB91] K. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 29(3):335–363, 1991.

[AB94] Krysztof R. Apt and Roland N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19-20:9–71, 1994.

[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, 1988.

[ADN97] Chandrabose Aravindan, Jürgen Dix, and Ilkka Niemelä. The DisLoP-project. Technical Report TR 1/97, University of Koblenz, Department of Computer Science, Rheinau 1, January 1997.

[ADP95] J. J. Alferes, C. V. Damásio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 14(1):93–147, 1995.

[AP96] Jose Julio Alferes and Luiz Moniz Pereira, editors. *Reasoning with Logic Programming*, LNAI 1111, Berlin, 1996. Springer.

[APP96] J. J. Alferes, L. M. Pereira, and T. Przymusinski. “Classical” negation in non monotonic reasoning and logic programming. In H. Kautz and B. Selman, editors, *4th Int. Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, USA, January 1996. Florida Atlantic University.

[BD93] Roland N. Bol and L. Degerstedt. Tabulated resolution for well–founded semantics. In *Proc. Int. Logic Programming Symposium'93*, Cambridge, Mass., 1993. MIT Press.

[12] `ftp://ftp.informatik.uni-hannover.de/software/index.html`

[13] `ftp://ftp.informatik.uni-hannover.de/software/static/static.html`

[BD94] Stefan Brass and Jürgen Dix. A disjunctive semantics based on unfolding and bottom-up evaluation. In Bernd Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen,* (IFIP '94-Congress, Workshop FG2: Disjunctive Logic Programming and Disjunctive Databases), pages 83–91, Berlin, 1994. Springer.

[BD95] Stefan Brass and Jürgen Dix. A General Approach to Bottom-Up Computation of Disjunctive Semantics. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 127–155. Springer, Berlin, 1995.

[BD96a] Stefan Brass and Jürgen Dix. Characterizing D-WFS: Confluence and Iterated GCWA. In L.M. Pereira J.J. Alferes and E. Orlowska, editors, *Logics in Artificial Intelligence (JELIA '96)*, LNCS 1126, pages 268–283. Springer, 1996. (Extended version will appear in the *Journal of Automated Reasoning* in 1997.).

[BD96b] Gerhard Brewka and Jürgen Dix. Knowledge representation with logic programs. Technical report, Tutorial Notes of the 12th European Conference on Artificial Intelligence (ECAI '96), 1996. Also appeared as Technical Report 15/96, Dept. of CS of the University of Koblenz-Landau. Will appear as Chapter 6 in *Handbook of Philosophical Logic*, 2nd edition (1998), Volume 6, Methodologies.

[BD97a] Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. *Journal of Logic Programming*, forthcoming, 1997. (Extended abstract appeared in: Characterizations of the Stable Semantics by Partial Evaluation *LPNMR, Proceedings of the Third International Conference, Kentucky*, pages 85–98, 1995. Springer.).

[BD97b] Stefan Brass and Jürgen Dix. Semantics of Disjunctive Logic Programs Based on Partial Evaluation. *Journal of Logic Programming*, accepted for publication, 1997. (Extended abstract appeared in: Disjunctive Semantics Based upon Partial and Bottom-Up Evaluation, *Proceedings of the 12-th International Logic Programming Conference, Tokyo*, pages 199–213, 1995. MIT Press.).

[BDK97] Gerd Brewka, Jürgen Dix, and Kurt Konolige. *Nonmonotonic Reasoning: An Overview.* CSLI Lecture Notes 73. CSLI Publications, Stanford, CA, 1997.

[BDNP97] Stefan Brass, Jürgen Dix, Ilkka Niemelä, and Teodor. C. Przymusinski. Comparison and Efficient Computation of the Static and the Disjunctive WFS. Technical report, University of Koblenz, Department of Computer Science, Rheinau 1, January 1997. submitted to a conference. Preliminary version appeared as Technical Report 2/96.

[BDP96] Stefan Brass, Jürgen Dix, and Teodor. C. Przymusinski. Super Logic Programs. In L. C. Aiello, J. Doyle, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR '96)*, pages 529–541. San Francisco, CA, Morgan Kaufmann, 1996.

[BF88] N. Bidoit and C. Froidevaux. General logic databases and programs: default logic semantics and stratification. *Journal of Information and Computation*, 1988.

[BF94] P. Baumgartner and U. Furbach. Model Elimination without Contrapositives and its Application to PTTP. *Journal of Automated Reasoning*, 13:339–359, 1994. Short version in: Proceedings of CADE-12, Springer LNAI 814, 1994, pp 87–101.

[BFN96] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper tableaux. In L.M. Pereira J.J. Alferes and E. Orlowska, editors, *Logics in Artificial Intelligence (JELIA '96)*, LNCS 1126, pages 1–17. Springer, 1996.

[BL93] Stefan Brass and Udo W. Lipeck. Bottom-up query evaluation with partially ordered defaults. In Stefano Ceri, Katsumi Tanaka, and Shalom Tsur, editors, *Deductive and Object-Oriented Databases, Third Int. Conf., (DOOD'93)*, number 760 in LNCS, pages 253–266, Berlin, 1993. Springer.

[BLM90] Chitta Baral, Jorge Lobo, and Jack Minker. Generalized Disjunctive Well-founded Semantics for Logic Programs: Procedural Semantics. In Z.W. Ras, M. Zemankova, and M.L Emrich, editors, *Proceedings of the 5th Int. Symp. on Methodologies for Intelligent Systems, Knoxville, TN, October 1990*, pages 456–464. North-Holland, 1990.

[BNNS93] Colin Bell, Anil Nerode, Raymond T. Ng, and V. S. Subrahmanian. Implementing Stable Semantics by Linear Programming. In Luis Moniz Pereira and Anil Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Second International Workshop*, pages 23–42, Cambridge, Mass., July 1993. Lisbon, MIT Press.

[BNNS94] Colin Bell, Anil Nerode, Raymond T. Ng, and V. S. Subrahmanian. Mixed Integer Programming Methods for Computing Non-Monotonic Deductive Databases. *Journal of the ACM*, 41(6):1178–1215, November 1994.

[Bry89] François Bry. Logic programming as constructivism: A formalization and its application to databases. In *Proc. of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'89)*, pages 34–50, 1989.

[BS91] C. Baral and V. S. Subrahmanian. Dualities between alternative semantics for logic programming and nonmonotonic reasoning. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *LP & NMR*, pages 69–86. MIT Press, 1991.

[CKPR73] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme-machine en français. Technical report, Groupe de Intelligence Artificielle Universite de Aix-Marseille II, 1973.

[CL95] Stefania Costantini and Gaetano A. Lanzarone. Static Semantics as Program Transformation and Well-founded Computation. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 156–180. Springer, Berlin, 1995.

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

[CSW95] Weidong Chen, Terrance Swift, and David S. Warren. Efficient Top-Down Computation of Queries under the Well-Founded Semantics. *Journal of Logic Programming*, 24(3):219–245, 1995.

[CW93] Weidong Chen and David S. Warren. A Goal Oriented Approach to Computing The Well-founded Semantics. *Journal of Logic Programming*, 17:279–300, 1993.

[CW95] Weidong Chen and David S. Warren. Computing of Stable Models and its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 17:279–300, 1995.

[CW96] Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[Dam96] C. V. Damásio. *Paraconsistent Extended Logic Programming with Constraints*. PhD thesis, Universidade Nova de Lisboa, October 1996.

[DF96] J. Dix and U. Furbach. The DFG-Project DisLoP on Disjunctive Logic Programming. *Computational Logic*, 2:89–90, 1996.

[DFN97] J. Dix, U. Furbach, and A. Nerode, editors. *Logic Programming and Nonmonotonic Reasoning*, LNAI to appear, Berlin, 1997. Springer.

[DHSH96] Roy Dyckhoff, Heinrich Herre, and Peter Schroeder-Heister, editors. *Extensions of Logic Programming*, LNAI 1050, Berlin, 1996. Springer.

[Dix91] J. Dix. Classifying semantics of logic programs. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *LP & NMR*, pages 166–180. MIT Press, 1991.

[Dix92a] J. Dix. A framework for representing and characterizing semantics of logic programs. In B. Nebel, C. Rich, and W. Swartout, editors, *3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1992.

[Dix92b] Jürgen Dix. Classifying Semantics of Disjunctive Logic Programs. In K. R. Apt, editor, *LOGIC PROGRAMMING: Proceedings of the 1992 Joint International Conference and Symposium*, pages 798–812, Cambridge, Mass., November 1992. MIT Press.

[Dix95a] Jürgen Dix. A Classification-Theory of Semantics of Normal Logic Programs: I. Strong Properties. *Fundamenta Informaticae*, XXII(3):227–255, 1995.

[Dix95b] Jürgen Dix. A Classification-Theory of Semantics of Normal Logic Programs: II. Weak Properties. *Fundamenta Informaticae*, XXII(3):257–288, 1995.

[Dix95c] Jürgen Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In Andre Fuhrmann and Hans Rott, editors, *Logic, Action and Information – Essays on Logic in Philosophy and Artificial Intelligence*, pages 241–327. DeGruyter, 1995.

[DLMW96] Jürgen Dix, Donald Loveland, Jack Minker, and David. S. Warren. Disjunctive Logic Programming and databases: Nonmonotonic Aspects. Technical Report Dagstuhl Seminar Report 150, IBFI GmbH, Schloß Dagstuhl, 1996.

[DM92] Jürgen Dix and Martin Müller. Abstract Properties and Computational Complexity of Semantics for Disjunctive Logic Programs. In *Proc. of the Workshop W1, Structural Complexity and Recursion-theoretic Methods in Logic Programming, following the JICSLP '92*, pages 15–28. H. Blair and W. Marek and A. Nerode and J. Remmel, November 1992. also available as Technical Report 13/93, University of Koblenz, Department of Computer Science.

[DM93] Jürgen Dix and Martin Müller. Implementing Semantics for Disjunctive Logic Programs Using Fringes and Abstract Properties. In Luis Moniz Pereira and Anil Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Second International Workshop*, pages 43–59, Cambridge, Mass., July 1993. Lisbon, MIT Press.

[DN95] Lars Degerstedt and Ulf Nilsson. Magic Computation of Well-founded Semantics. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 181–204. Springer, Berlin, 1995.

[DPP95] J. Dix, L. Pereira, and T. Przymusinski, editors. *Non-Monotonic Extensions of Logic Programming*, LNAI 927, Berlin, 1995. Springer.

[Dun91] P. M. Dung. Negation as hypotheses: An abductive framework for logic programming. In K. Furukawa, editor, *8th Int. Conf. on LP*, pages 3–17. MIT Press, 1991.

[EG93] Thomas Eiter and Georg Gottlob. Propositional Circumscription and Extended Closed World Reasoning are Π_2^P-complete. *Theoretical Computer Science*, 144(2):231–245, Addendum: vol. 118, p. 315, 1993, 1993.

[EGLS96] T. Eiter, G. Gottlob, J. Lu, and V. S. Subrahmanian. Computing Non-Ground Representations of Stable Models. Technical report, University of Maryland, 1996.

[EGM93] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Expressive Power and Complexity of Disjunctive DATALOG. In *Proceedings of Workshop on Logic Programming with Incomplete Information, Vancouver Oct. 1993, following ILPS' 93*, pages 59–79, 1993.

[EGM94] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Adding disjunction to datalog. In *Proc. of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'94)*, pages 267–278, 1994.

[EK76] M. Van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 4(23):733–742, 1976.

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

[FLMS93] J. A. Fernández, J. Lobo, J. Minker, and V.S. Subrahmanian. Disjunctive LP + Integrity Constraints = Stable Model Semantics. *Annals of Mathematics and Artificial Intelligence*, 8(3-4), 1993.

[FM95] J. A. Fernández and J. Minker. Bottom-Up Computation of Perfect Models for Disjunctive Theories. *Journal of Logic Programming*, 25(1):33–51, 1995.

[Gel87] M. Gelfond. On stratified autoepistemic theories. In *AAAI'87*, pages 207–211. Morgan Kaufmann, 1987.

[Gel89a] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *8th Symposium on Principles of Database Systems*. ACM SIGACT-SIGMOD, 1989.

[Gel89b] A. Van Gelder. Negation as failure using tight derivations for general logic programs. *Journal of LP*, 6(1):109–133, 1989.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th Int. Conf. on LP*, pages 1070–1080. MIT Press, 1988.

[GL89] Michael Gelfond and Vladimir Lifschitz. Compiling Circumscriptive Theories into Logic Programs. In Reinfrank, de Kleer, Ginsberg, and Sandewall, editors, *Non-Monotonic Reasoning*, LNAI 346, pages 74–99, Berlin, January 1989. Springer.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–387, 1991. (Extended abstract appeared in: Logic Programs with Classical Negation. *Proceedings of the 7-th International Logic Programming Conference, Jerusalem*, pages 579-597, 1990. MIT Press.).

[GL92] M. Gelfond and V. Lifschitz. Representing actions in extended logic programs. In K. Apt, editor, *Int. Joint Conf. and Symp. on LP*, pages 559–573. MIT Press, 1992.

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

[GRS91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[HMS81] HMSO. *British Nationality Act.* Her Majesty's Stationery Office, 1981.

[IKH92] Katsumi Inoue, M. Koshimura, and R. Hasegawa. Embedding negation-as-failure into a model generation theorem prover. In Deepak Kapur, editor, *Automated Deduction — CADE-11*, number 607 in LNAI, Berlin, 1992. Springer.

[Ino91] K. Inoue. Extended logic programs with default assumptions. In Koichi Furukawa, editor, *8th Int. Conf. on LP*, pages 490–504, Cambridge, Mass., 1991. MIT Press.

[KK71] R. Kowalski and D. Khuener. Linear resolution with selection function. *Artificial Intelligence*, 5:227–260, 1971.

[KNS94] Vadim Kagan, Anil Nerode, and V. S. Subrahmanian. Computing Definite Logic Programs by Partial Instantiation. *Annals of Pure and Applied Logic*, 67:161–182, 1994.

[KNS95] Vadim Kagan, Anil Nerode, and V. S. Subrahmanian. Computing Minimal Models by Partial Instantiation. *Theoretical Computer Science*, 155:157–177, 1995.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In *Proceeedings IFIP' 74*, pages 569–574. North Holland Publishing Company, 1974.

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

[Kow89] R. Kowalski. The treatment of negation in logic programs for representing legislation. In *2nd Int. Conf. on AI and Law*, pages 11–15, 1989.

[Kow90] R. Kowalski. Problems and promises of computational logic. In John W. Lloyd, editor, *Computational Logic*, Basic Research Series, pages 1–36, Berlin, 1990. Springer.

[Kow92] R. Kowalski. Legislation as logic programs. In *Logic Programming in Action*, pages 203–230. Springer–Verlag, 1992.

[KS90] R. Kowalski and F. Sadri. Logic programs with exceptions. In Warren and Szeredi, editors, *7th Int. Conf. on LP*, Cambridge, Mass., 1990. MIT Press.

[KSS91] David B. Kemp, Peter J. Stuckey, and Divesh Srivastava. Magic Sets and Bottom-Up Evaluation of Well-Founded Models. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 Int. Symposium on Logic Programming*, pages 337–351. MIT, June 1991.

[Kun87] Kenneth Kunen. Negation in Logic Programming. *Journal of Logic Programming*, 4:289–308, 1987.

[Kun90] Kenneth Kunen. Some Remarks on the completed Database. *Fundamenta Informaticae*, XIII:35–49, 1990.

[Lev86] H. Levesque. Making believers out of computers. *Artificial Intelligence*, 30:81–107, 1986.

[Lip79] W. Lipski, Jr. On semantic issues connected with incomplete information databases. *ACM Transactions on Database Systems*, 4:262–296, 1979.

[Llo87] John W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 1987. 2nd edition.

[LMR92] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming.* MIT-Press, 1992.

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

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

[Min88] Jack Minker. *Foundations of Deductive Databases.* Morgan Kaufmann, 95 First Street, Los Altos, CA 94022, 1st edition, 1988.

[Min93] Jack Minker. An Overview of Nonmonotonic Reasoning and Logic Programming. *Journal of Logic Programming, Special Issue*, 17, 1993.

[Min96] Jack Minker. Logic and databases: A 20 year retrospective. In Dino Pedreschi and Carlo Zaniolo, editors, *Proceedings of the International Workshop on Logic in Databases (LID)*, LNCS 1154, pages 3–58. Springer, Berlin, 1996.

[MNT95] W. Marek, A. Nerode, and M. Truszczyński, editors. *Logic Programming and Nonmonotonic Reasoning*, LNAI 928, Berlin, 1995. Springer.

[Mon92] L. Monteiro. Notes on the negation in logic programs. Technical report, Dep. of Computer Science, Univerdade Nova de Lisboa, 1992. Course Notes, 3rd Advanced School on AI, Azores, Portugal, 1992.

[Moo85] R. Moore. Semantics considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[MR95] Jack Minker and Carolina Ruiz. Computing stable and partial stable models of extended disjunctive logic programs. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 205–229. Springer, Berlin, 1995.

[MT91] W. Marek and M. Truszczynski. Autoepistemic logics. *Journal of the ACM*, 38(3):588–619, 1991.

[Mül92] Martin Müller. Examples and Run-Time Data from KORF, 1992.

[Nel49] D. Nelson. Constructible falsity. *JSL*, 14:16–26, 1949.

[NMS91] A. Nerode, W. Marek, and V. S. Subrahmanian, editors. *Logic Programming and Non–monotonic Reasoning: Proceedings of the First Int. Ws.*, Washington D.C., USA, 1991. The MIT Press.

[NNS91] Anil Nerode, Raymond T. Ng, and V.S. Subrahmanian. Computing Circumscriptive Deductive Databases. CS-TR 91-66, Computer Science Dept., Univ. Maryland, University of Maryland, College Park, Maryland, 20742, USA, December 1991.

[NS96] Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, September 1996. The MIT Press.

[PA92] L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *European Conf. on AI*, pages 102–106. John Wiley & Sons, 1992.

[PAA91a] L. M. Pereira, J. J. Alferes, and J. N. Aparício. A practical introduction to well founded semantics. In B. Mayoh, editor, *Scandinavian Conf. on AI.* IOS Press, 1991.

[PAA91b] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Counterfactual reasoning based on revising assumptions. In Ueda and Saraswat, editors, *Int. LP Symp.*, pages 566–577. MIT Press, 1991.

[PAA91c] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Nonmonotonic reasoning with well founded semantics. In Koichi Furukawa, editor, *8th Int. Conf. on LP*, pages 475–489. MIT Press, 1991.

[PAA92] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Logic programming for nonmonotonic reasoning. In *Applied Logic Conf.* Preproceedings by ILLC, Amsterdam, 1992. To appear in Springer–Verlag LNAI.

[PAA93] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Non-Monotonic Reasoning with Logic Programming. *Journal of Logic Programming*, 17:227–264, 1993.

[PDA93] L. M. Pereira, C. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal. In L. M. Pereira and A. Nerode, editors, *2nd Int. Ws. on LP & NMR*, pages 316–330, Cambridge, Mass., 1993. MIT Press.

[PN93] L. M. Pereira and A. Nerode, editors. *Logic Programming and Non–monotonic Reasoning: Proceedings of the Second Int. Ws.*, Lisboa, Portugal, 1993. MIT Press.

[PP88] H. Przymusinska and T. Przymusinski. Weakly perfect model semantics. In R. Kowalski and K. A. Bowen, editors, *5th Int. Conf. on LP*, pages 1106–1122. MIT Press, 1988.

[PP90] H. Przymusinska and T. Przymusinski. Semantic issues in deductive databases and logic programs. In R. Banerji, editor, *Formal Techniques in AI, a Sourcebook*, pages 321–367. North Holland, 1990.

[Prz89a] T. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *8th Symp. on Principles of Database Systems*. ACM SIGACT-SIGMOD, 1989.

[Prz89b] T. Przymusinski. Three–valued non–monotonic formalisms and logic programming. In R. Brachman, H. Levesque, and R. Reiter, editors, *1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 341–348. Morgan Kaufmann, 1989.

[Prz89c] Teodor Przymusinski. On the declarative and procedural Semantics of logic Programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[Prz90] T. Przymusinski. Extended stable semantics for normal and disjunctive programs. In Warren and Szeredi, editors, *7th Int. Conf. on LP*, pages 459–477, Cambridge, Mass., 1990. MIT Press.

[Prz91a] T. Przymusinski. Autoepistemic logic of closed beliefs and logic programming. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *LP & NMR*, pages 3–20. MIT Press, 1991.

[Prz91b] Teodor 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, Cambridge, Mass.).

[Prz91c] Teodor Przymusinski. Stationary Semantics for Normal and Disjunctive Logic Programs. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *DOOD '91, Proceedings of the 2nd International Conference*, Berlin, December 1991. Muenchen, Springer. LNCS 566.

[Prz95a] Teodor Przymusinski. Semantics of normal and disjunctive logic programs: A unifying framework. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Proceedings of the Workshop on Non-Monotonic Extensions of Logic Programming at the Eleventh International Logic Programming Conference, ICLP'94, Santa Margherita Ligure, Italy, June 1994*, pages 43–67. Springer, 1995.

[Prz95b] Teodor Przymusinski. Static Semantics For Normal and Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 14:323–357, 1995.

[PW90] D. Pearce and G. Wagner. Reasoning with negative information I: Strong negation in logic programs. In L. Haaparanta, M. Kusch, and I. Niiniluoto, editors, *Language, Knowledge and Intentionality*, pages 430–453. Acta Philosophica Fennica 49, 1990.

[Rei78a] R. Reiter. On closed–world data bases. In H. Gallaire and J. Minker, editors, *Logic and DataBases*, pages 55–76. Plenum Press, 1978.

[Rei78b] Raymond Reiter. On closed world data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76, New York, 1978. Plenum.

[Rei80] Raymond Reiter. A Logic for Default-Reasoning. *Artificial Intelligence*, 13:81–132, 1980.

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

[RN95] Stuart Russel and Peter Norvig. *Artificial Intelligence — A Modern Approach.* Prentice Hall, New Jersey 07458, 1995.

[Ros92] Kenneth A. Ross. A procedural semantics for well-founded negation in logic programs. *Journal of Logic Programming*, 13:1–22, 1992.

[RT88] Kenneth A. Ross and Rodney A. Topor. Inferring negative Information from disjunctive Databases. *Journal of Automated Reasoning*, 4:397–424, 1988.

[Sak89] Chiaki Sakama. Possible Model Semantics for Disjunctive Databases. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Deductive and Object-Oriented Databases, Proceedings of the First International Conference (DOOD89)*, pages 1055–1060, Kyoto, Japan, 1989. North-Holland Publ.Co.

[She88] John C. Shepherdson. Negation in Logic Programming. In Jack Minker, editor, *Foundations of Deductive Databases*, chapter 1, pages 19–88. Morgan Kaufmann, 1988.

[She90] J. Shepherdson. Negation as failure, completion and stratification. In *Handbook of AI and LP*, 1990.

[SI93] Chiaki Sakama and Katsumi Inoue. Negation in Disjunctive Logic Programs. In D. Warren and Peter Szeredi, editors, *Proceedings of the 10th Int. Conf. on Logic Programming, Budapest*, Cambridge, Mass., July 1993. MIT Press.

[SS95] Chiaki Sakama and Hirohisa Seki. Partial Deduction of Disjunctive Logic Programs: A Declarative Approach. In *Logic Program Synthesis and Transformation – Meta Programming in Logic*, LNCS 883, pages 170–182, Berlin, 1995. Springer. Extended version to appear in *Journal of Logic Programming.*

[TS86] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proceedings of the Third International Conference on Logic Programming, London*, LNAI, pages 84–98, Berlin, June 1986. Springer.

[Wag91] G. Wagner. A database needs two kinds of negation. In B. Thalheim, J. Demetrovics, and H-D. Gerhardt, editors, *Mathematical Foundations of Database Systems*, LNCS 495, pages 357–371, Berlin, 1991. Springer.

[Wol94] Bernd Wolfinger, editor. *GI-Fachgespräch 2:* Disjunktive logische Programmierung und disjunktive Datenbanken, pages 51–100. Springer, Berlin, 1994.

[WPP77] D. H. Warren, L. M. Pereira, and F. Pereira. Prolog: The language and its implementation compared with Lisp. In *Symp. on AI and Programming Languages*, pages 109–115. ACM SIGPLAN–SIGART, 1977.