Logic Programming and Knowledge Representation

Chitta Baral and Michael Gelfond
Computer Science Department
University of Texas at El Paso
El Paso, Texas 79968
{chitta,mgelfond}@cs.ep.utexas.edu

January 24, 1994

Abstract

In this paper, we review recent work aimed at the application of declarative logic programming to knowledge representation in artificial intelligence. We consider extensions of the language of definite logic programs by classical (strong) negation, disjunction, and some modal operators and show how each of the added features extends the representational power of the language.

We also discuss extensions of logic programming allowing abductive reasoning, meta-reasoning and reasoning in open domains. We investigate the methodology of using these languages for representing various forms of nonmonotonic reasoning and for describing knowledge in specific domains. We also address recent work on properties of programs needed for sucessful applications of this methodology such as consistency, categoricity and complexity.

Contents

1 Introduction

In this paper, we review recent work aimed at the application of logic programming to knowledge representation in artificial intelligence (AI). We consider various extensions of "pure Prolog" (definite logic programs) and show how each of the added features extends the representational power of the language.

1.1 Historical Perspective

Knowledge representation is one of the most important subareas of artificial intelligence. If we want to design an entity (a machine or a program) capable of behaving intelligently in some environment, then we need to supply this entity with sufficient knowledge about this environment. To do that, we need an unambiguous language capable of expressing this knowledge, together with some precise and well understood way of manipulating sets of sentences of the language which will allow us to draw inferences, answer queries, and to update both the knowledge base and the desired program behavior.

Around 1960, McCarthy [McC59] first proposed the use of *logical formulas* as a basis for a knowledge representation language of this type. This is how he explains the advantages of such a representation:

> Expressing information in declarative sentences is far more modular than expressing it in segments of computer programs or in tables. Sentences can be true in a much wider context than specific programs can be used. The supplier of a fact does not have to understand much about how the receiver functions or how or whether the receiver will use it. The same fact can be used for many purposes, because the logical consequences of collections of facts can be available.

This idea has been further developed by many researchers with various backgrounds and interests. First, the classical logic of predicate calculus served as the main technical tool for the representation of knowledge. It has a well-defined semantics and a well-understood and powerful inference mechanism, and it proved to be sufficiently expressive for the representation of mathematical knowledge. It was soon realized, however, that for the representation of *commonsense knowledge*, this tool is inadequate. The difficulty is rather deep and related to the so-called "monotonicity" of theories based on predicate calculus. A logic is called *monotonic* if the addition of new axioms to a theory based on it never leads to the loss of any theorems proved in this theory. Commonsense reasoning is nonmonotonic: new information constantly forces us to withdraw previous conclusions. This observation has led to the development and investigation of new logical formalisms, *nonmonotonic logics*. The best known of them are circumscription [McC80, McC86, Lif85b], default logic [Rei80b], and nonmonotonic modal logics [MD80, McD82, Moo85]. A collection of important papers on nonmonotonic reasoning published before 1987 appears in [Gin87]. A survey can be found in [Rei87a]. Much technical work has been done to investigate the mathematical properties of these logics, as well as their applicability to the formalization of commonsense reasoning in various specific domains. This work has substantially deepened our understanding of the properties of nonmonotonic reasoning and of the technical problems involved in its formalization.

Another direction of research, started by Green [Gre69], Hayes [Hay73] and Kowalski [Kow74], and continued by many others, combined the idea of logic as a representation language with the theory of automated deduction and constructive logic. This led Kowalski and Colmerauer to the creation of *logic programming* [Llo87] and the development of the first logic programming language, Prolog [CKPR73].

Even though logic programming and nonmonotonic logic share many common goals and techniques, until recently, there were no strong ties between the two research communities. Originally, declarative Prolog was defined as a small subset of predicate calculus. This dialect of Prolog is now called "pure" Prolog. The restricted syntax of pure Prolog makes it possible to efficiently organize the process of inference, while its semantics relies heavily on the classical, model-theoretic notion of logical entailment. Unlike nonmonotonic logics, with their emphasis on expressiveness, efficiency and development of programming methodology seemed to be the main concern of the logic programming community.

With time, however, Prolog evolved to incorporate some nonclassical, nonmonotonic features, which made it closer in spirit to the nonmonotonic logics mentioned above. The most important nonmonotonic feature of modern Prolog is *negation as failure* [Cla78, Rei78]. The initial definition of this construct was purely procedural, which inhibited its use for knowledge representation and software engineering, as well as for the investigation of the relationship between logic programming and other nonmonotonic formalisms. Work, started by Clark and Reiter in the late 70's, was aimed at the development of a declarative semantics for logic programs with negation as failure.

The problem proved to be a rather nontrivial one. After more than ten years of extensive investigation, we now have a much better understanding of the problems involved, but there is still no universally accepted semantics for logic programs with negation as failure, even though for large classes of programs a certain level of consensus seems to have been achieved. The work on the declarative semantics of negation as failure has significantly enhanced our understanding of the relationship between nonmonotonic logics and logic programming.

It became apparent, on the other hand, that in order to become satisfactory tools for knowledge representation, logic programming languages should be expanded to allow for better handling of incomplete information. Work in this direction was started by Minker [Min82], Loveland [Lov87] and others, who investigated the possibility of expanding logic programs by *disjunctive information.* In [GL90, PW89, Gel92b], extensions of logic programming by *classical* (or *strong*) *negation* and epistemic operators were suggested. Unlike "traditional" nonmonotonic formalisms, these extensions are not based on the use of classical logical connectives, and do not include full first-order logic (not even its propositional part). Their fairly simple syntactic form may facilitate the adaptation of query–answering methods developed in the context of logic programming and deductive databases to more complicated forms of knowledge representation and reasoning. At the same time, these logic programming-based languages are rather expressive. In fact, they are more expressive than first-order logic. (See Section 10.)

¿From the perspective of knowledge representation, such extensions of traditional logic programming have the same status as other nonmonotonic formalisms and should be studied as such. This includes the investigation of the methodology of using these languages for representing various forms of nonmonotonic reasoning and for describing knowledge in specific domains; the mathematical investigation of properties of theories stated in these languages, done from the stand point of their semantics and not necessarily related to any particular computational mechanism; development of query answering systems; and investigation of the relationship between logic programming and other knowledge representation methods.

1.2 Structure of the Paper

In this paper, we discuss some recent work in logic programming which contributes to this view. We will not write a comprehensive survey of the field; the paper will reflect the authors' views on what is important, with their preferences and biases, which explains the inclusion of large parts of the authors' own work. This paper does not contain any new mathematical result with the single exception of Proposition 4.1. Many important developments are omitted simply because of space and time limitation and/or inability of the authors to incorporate them in the whole picture. We hope, however, that it will allow the readers to feel the flavor of the problems involved in using logic programming for knowledge representation.

The rest of the paper is organized as follows. In Section 2 we consider general logic programs (also known as normal logic programs) and show how general logic programs can be used to represent knowledge in AI. In particular we consider McCarthy's [McC59] example of flying birds and the Yale shooting problem [HM87] and show their formalization using general logic programs. We also discuss formalization of normative statements[1] of the kind "A's are normally B's" using general logic programs. Our discussion is based on the stable model semantics of general logic programs. We briefly discuss the other semantics of general logic programs and discuss classes of general logic programs where the various semantics agree. We also review a method of computing the stable models of a general logic program.

In Section 3 we consider extended logic programs [GL91, Wag91] that allow classical negation (also referred to as "strong negation") and discuss its expressibility in the context of knowledge representation. We reformalize McCarthy's example of flying birds and the Yale shooting problem using extended logic programs and show the utility of using extended logic programs in the presence of incomplete information where the *closed world assumption* (CWA) [Rei78] can not be automatically assumed.

In Section 4 we consider disjunctive logic programs where disjunctions are allowed in the heads of the rules of the program. We formalize two examples from the literature. In particular we consider an example from [Poo89] that was used to demonstrate the difficulties associated with representing disjunctive information in Reiter's default logic. We also discuss other semantics of general logic programs and review a method to compute the answer set semantics of a disjunctive logic program.

In Section 5 we show the inadequacy of disjunctive logic programs in representing certain kinds of information and introduce two new unary operators K (meaning *known*) and M (meaning *may be believed*). The extension of disjunctive logic programs by these operators (called epistemic logic programs) is used to overcome this inadequacy.

[1] Normative or normic statements [Scr59, Scr63] frequently involve terms such as 'naturally', 'normally', 'typically', 'tendency', 'ought', 'should' and other.

In Section 6 we consider the framework of meta-logic programming and discuss several of its features. In particular we discuss an application of results related to logic programming semantics to proving correctness of simple meta-interpreters for logic programs. We also consider several meta-logic programs that formalize database updates and hypothetical reasoning.

In Section 7 we discuss the modification of the semantics of logic programs and disjunctive databases which allows for *reasoning in the absence of the domain–closure assumption [Rei80a]*. This modification increases the expressive power of the language and allows one to explicitly state the domain–closure and other assumptions about the domain of discourse in the language of logic programming.

In Section 8 we discuss a logic programming language based on abduction. We then discuss abduction as a formalism for explanation of observations and describe the connection between abduction and negation as failure.

In Section 9 we discuss the relationship between the logic programming–based formalisms discussed in the previous sections and various nonmonotonic logics developed in artificial intelligence, such as circumscription, default logic, autoepistemic logic and truth maintenance systems.

In Section 10 we discuss the complexity and expressibility of logic programming languages and in Section 11 we conclude by mentioning some further problems that need to be addressed.

2 General Logic Programs

2.1 Preliminaries

The language of a logic program, like a first-order language, is determined by its object constants, function constants and predicate constants. Terms are built as in the corresponding first-order language; atoms have the form $p(t_1, \ldots, t_n)$, where the t's are terms and p is a predicate symbol of arity n.. A rule is an expression of the form

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \tag{1}$$

where A_i's are atoms and *not* is a logical connective called *negation as failure* [Cla78, Rei78]. The left-hand side of the rule is called the rule's head or conclusion; the right-hand side is called the rule's body (or premise). A collection of rules is called a *general logic program*. (They are also referred to as *normal logic programs*.) General logic programs that do not have *not* are called *definite programs*. Formulas and rules not containing variables are called *ground*. The set of all ground atoms in the language of a program Π will be denoted by $HB(\Pi)$ (Herbrand base of Π) with Π omitted whenever possible. For a predicate p, $atoms(p)$ will denote the subset of $HB(\Pi)$ formed with predicate p and for a set of predicates A, $atoms(A)$ will denote the subset of $HB(\Pi)$ formed with the predicates in A. Unless otherwise stated, we assume that rules with variables (usually denoted by capital letters) are used as shorthand for the sets of all their ground instantiations.

A logic program can be viewed as a specification for building possible theories of the world, and the rules can be viewed as constraints these theories should satisfy. Semantics of logic programs differ in the way they define satisfiability of the rules. In this paper we will mainly use the *stable model* semantics [GL88] and its extensions but most of our discussion will be semantics independent. Under this semantics the corresponding theories are sets of ground atoms, called the *stable models* of a program. They are defined as follows:

Definition 2.1 The *stable model* of a definite program Π is the smallest subset S of HB such that for any rule $A_0 \leftarrow A_1, \ldots, A_m$ from Π, if $A_1, \ldots, A_m \in S$, then $A_0 \in S$.

The stable model of a definite program Π is denoted by $a(\Pi)$.

Let Π be an arbitrary general logic program. For any set S of atoms, let Π^S be a program obtained from Π by deleting

(i) each rule that has a formula *not* A in its body with $A \in S$, and

(ii) all formulas of the form *not* A in the bodies of the remaining rules.

Clearly, Π^S does not contain *not*, so that its stable model is already defined. If this stable model coincides with S, then we say that S is a *stable model* of Π. In other words, a stable model of Π is characterized by the equation

$$S = a(\Pi^S). \tag{2}$$

□

A ground atom P is *true* in S if $P \in S$, otherwise P is *false* (i.e. $\neg P$ is *true*) is S. The definition is extended to arbitrary first-order formulas in the standard way. Π *entails* a formula f ($\Pi \models f$) if f is *true* in all stable models of Π. We will say that the answer to a ground query q is *yes* if q is *true* in all stable models of Π (i.e. $\Pi \models q$), *no* if $\neg q$ is *true* in all stable models of Π (i.e. $\Pi \models \neg q$) and *unknown* otherwise.

Example 2.1 Assume that our language contains two object constants a and b and consider

$\Pi = \{p(X) \leftarrow not\ q(X), \quad q(a) \leftarrow\}$

Let us show that a set $S = \{q(a), p(b)\}$ is a stable model of Π. By construction, $\Pi^S = \{p(b) \leftarrow, \quad q(a) \leftarrow\}$ whose stable model is obviously equal to S. Later we will show that there is no other stable model of Π. □

It is easy to see that logic programs are *nonmonotonic*, i.e. adding new information to the program may force a reasoner associated with it to withdraw its previous conclusions about the world. This happens for instance if we expand the program from Example 2.1 by a new fact $q(b) \leftarrow$. It is easy to see that the old program entails $p(b)$ while the new one does not.

The above notion of entailment can also be defined in terms of models of classical logic. To do that every rule in a program Π of the form (1) is replaced by the first-order formula

$A_1 \wedge \ldots \wedge A_m \wedge \neg A_{m+1} \wedge \ldots \neg A_n \supset A_0$.

The resulting first-order theory will be denoted by T_Π. If Π is definite then T_Π is a Horn theory and the stable model of Π coincide with the minimal (w.r.t. set-theoretic inclusion) Herbrand model of T_Π. (By Herbrand model of a first-order theory T we mean a collection of ground atoms that satisfies formulas from T). It is easy to show that a stable model of a general logic program Π is a model of its classical counterpart T_Π. This explains the use of the term "model" in the definition of the stable model semantics which was influenced by the "preferred models" [ABW88, Prz88a, VG88] approach to the semantics of logic programs. According to this approach a logic program Π is identified with its classical counterpart and its semantics is given in terms of some (preferred) class of models of Π. In many approaches to the semantics of logic programs they are still (consciously as well as often subconsciously) not separated from their classical counterparts. In extensions of logic programming discussed in this paper mapping of programs into classical theories becomes more complicated so we prefer to stress the non-classical character of the logic programming connectives from the beginning and to try to avoid the use of the term model in the Tarskian sense.

Uniqueness of a stable model is an important property of a program. Programs which have a unique stable model are called *categorical*.

The next two examples show that not all programs are categorical. There are programs with multiple stable models and with no stable models at all. The latter will be called *incoherent*. Programs with at least one stable model are called *coherent*.

Example 2.2 Consider the general logic program $\Pi = \{p \leftarrow not\ p\}$. We now show that it is incoherent. Let us assume that it has a stable model S. Consider two cases:

(a) if $p \in S$ then Π^S is empty and so is its stable model. Since S is not empty it is not a stable model of Π
(b) if $p \notin S$ then $\Pi^S = \{p \leftarrow\}$, its stable model is $\{p\}$ and hence S is not a stable model of Π. The contradiction falsifies our assumption and therefore Π has no stable models. $\square$

The program from the next example has two stable models.

Example 2.3 Consider a general logic program

$p \leftarrow not\ q$

$q \leftarrow not\ p.$

It is easy to check that this program has two stable models $\{p\}$ and $\{q\}$. $\square$

Coherence and categoricity are important properties of logic programs. There is a collection of results giving sufficient conditions for these properties. We will now discuss some of these results. We start with the class of stratified programs [ABW88, VG88, CH85].

Definition 2.2 A partition $\pi_0, \ldots, \pi_k$ of the set of all predicate symbols of a general logic program Π is *a stratification* of Π, if for any rule of the type (1) and for any $p \in \pi_s$, $0 \leq s \leq k$ if $A_0 \in atoms(p)$, then:
(a) for every $1 \leq i \leq m$ there is q and $j \leq s$ such that $q \in \pi_j$ and $A_i \in atoms(q)$
(b) for every $m+1 \leq i \leq n$ there is q and $j < s$ such that $q \in \pi_j$ and $A_i \in atoms(q)$.
i.e. $\pi_0, \ldots, \pi_k$ is a stratification of π if for all rules in π, the predicates that appear only positively in the body of a rule are in strata lower than or equal to the stratum of the predicate in the head of the rule, and the predicates that appear under negation as failure are in strata lower than the stratum of the predicate in the head of the rule.

This stratification of the predicates defines a stratification of the rules to strata $\Pi_0, \ldots, \Pi_k$ where a strata Π_i contains rules whose heads are formed by predicates from π_i. Π_i can be viewed as a definition of relations from π_i. The above condition allows definitions which are mutually recursive but prohibits the use of negation as failure for the yet undefined predicates.

A program is called *stratified* if it has a stratification. $\square$

Example 2.4 A general logic program Π consisting of rules

$p(f(X)) \leftarrow p(X), not\ q(X)$

$p(a) \leftarrow$

$q(X) \leftarrow not\ r(X)$

$r(a) \leftarrow$

is stratified with a stratification $\{r\}, \{q\}, \{p\}$. $\square$

Given a program Π, the *dependency graph*, D_Π, of Π consists of the predicate names as the vertices and $< P_i, P_j, s >$ is a labeled edge in D_Π iff there is a rule r in Π with P_i in its head and P_j in its body and the label $s \in \{+, -\}$ denoting whether P_j appears in a positive or a negative literal in the body of r. Note that an edge may be labeled both by $+$ and $-$. A cycle in the dependency graph of a program is said to be a negative cycle if it contains at least one edge with a negative label.

Proposition 2.1 *[ABW88]* A general logic program Π is stratified iff its dependency graph D_Π does not contain any negative cycles. $\square$

The notion of stratification plays an important role in the fields of logic programming, deductive databases, and AI. The following theorem describes an important property of stratified programs.

Proposition 2.2 *[ABW88, GL88]* Any stratified general logic program is categorical. $\square$

It is easy to see that the program from Example 2.1 is stratified and therefore has only one stable model.

Existence of stable models was further studied in [Fag90, Cav89, Dun92, Prz88a]. The following result, due to Fages [Fag90], is representative of this direction of research. A general logic program is said to be *call-consistent* [Kun89, Sat87] if its dependency graph does not have a cycle with an odd number of negative edges.

Theorem 2.3 *[Fag90]* A call-consistent logic program whose dependency graph does not have a cycle with only positive edges has at least one stable model. $\square$

In our further discussion we will need the following Lemma about general logic programs.

Lemma 2.4 *[MS89]* For any stable model S of a general logic program Π:

(a) For any ground instance of a rule of the type (1) from Π,
if $\{A_1, \ldots, A_m\} \subseteq S$ and $\{A_{m+1}, \ldots, A_n\} \cap S = \emptyset$ then $A_0 \in S$.

(b) If S is a stable model of Π and $A_0 \in S$, then there exists a ground instance of a rule of the type (1) from Π such that
$\{A_1, \ldots, A_m\} \subseteq S$ and $\{A_{m+1}, \ldots, A_n\} \cap S = \emptyset$. $\square$

2.2 Representing Knowledge in General Logic Programs

In this section we discuss several examples of the use of general logic programs for representation of knowledge and for commonsense reasoning. We will start by demonstrating how general logic programs can be used to formalize **normative statements**, i.e., statements of the form "A's are **normally** (typically, as a rule, etc.) B's." Statements of this form are commonly used in various types of commonsense reasoning. The following story is due to McCarthy [McC59].

Suppose that a reasoning agent has the following knowledge about birds: birds typically fly and penguins are non-flying birds. He also knows that Tweety is a bird. Suppose now, that the agent is hired to build a cage for Tweety, and he leaves off the roof on the grounds that he does not know whether or not Tweety can fly. It would be reasonable for us to view this argument as invalid and to refuse the agent's product. This would not be the case if Tweety could not fly for some reason (unknown to the agent), and we refused to pay for the bird cage because the agent had "unnecessarily" put a roof on it. The following example shows how this type of knowledge can be represented by a general logic program.

Example 2.5 Consider a program[2] $\mathcal{B}$ consisting of the rules

$$\left.\begin{array}{l} 1.\ flies(X) \leftarrow bird(X), not\ ab(r1, X) \\ 2.\ bird(X) \leftarrow penguin(X) \\ 3.\ ab(r1, X) \leftarrow penguin(X) \\ 4.\ make\_top(X) \leftarrow flies(X) \end{array}\right\} \mathcal{B}$$

used together with some facts about particular birds, say,

f1. $bird(tweety) \leftarrow$

f2. $penguin(sam) \leftarrow$

Most predicate names in this example are self-explanatory. $r1$ is a constant in our language used to name the rule 1. and the atom $ab(r1, X)$ stands for birds whose flying ability is suspect (i.e. to which rule 1. is not applicable). The first rule expresses a normative statement about the flying ability of birds. (Statements of this sort are often called *default assumptions*, or just defaults.) It allows us to conclude that a bird X flies unless we can establish that it is exceptional with respect to flying. Rule 3., which is used to block the application of default 1. to penguins, is sometimes called a *cancellation rule*.

In general, normative statements of the form "a's are normally b's" are represented in the language of general logic programs by the rules

$$b(X) \leftarrow a(X), not\ ab(r, X) \tag{3}$$

where r is a constant of our language used to name the rule (3).

Similarly, the exception to a normative statement of the form "c's are exceptional a's. They are not b's" is represented by the rule

$$ab(r, X) \leftarrow c(X) \tag{4}$$

Exceptions of this sort will be called *strong exceptions*. (Compare with weak exceptions in (13) in Section 3.) The cancellation rule (4) can be viewed as a particular instance of a general reasoning principle called the *Inheritance Principle* [Tou86], according to which *more specific information is preferable to that which is more general.*

[2] We will discuss several versions of this program and use $\mathcal{B}$ with subscripts to denote the different versions.

It is easy to see that a general logic program $\mathcal{B}$ consisting of rules 1. – 4. and the facts (f1) and (f2) is stratified, and hence, has a unique stable model. Now let us use Lemma 2.4 to find answers to some queries about the flying abilities of various birds. We will start with the query $flies(tweety)$. Let S be the stable model of $\mathcal{B}$. By the lemma, $flies(tweety) \in S$ iff

(a) $bird(tweety) \in S$, and

(b) $ab(r1, tweety) \notin S$.

Statement (a) follows immediately from (f1) and the lemma. To prove (b), we need to show that $penguin(tweety) \notin S$, which follows immediately from the same lemma.

Hence, using (a) and (b), together with rule 1. and the first part of the lemma we have $flies(tweety) \in S$ and hence the answer to the query $flies(tweety)$ is *yes*. It is equally easy to show that the answer to the query $flies(sam)$ is *no*. □

The above example is a typical example of reasoning with inheritance hierarchies. We will use it throughout the paper. There is a vast literature on representing inheritance hierarchies using nonmonotonic formalisms (For a survey see [Hor]). Representing inheritance hierarchies using logic programs is discussed in [GP90, Lin91, PAA, PAA91c].

We conclude this section with a brief discussion of the application of general logic programs to the formalization of reasoning about results of actions. Let us start with a form of such reasoning called temporal projection, in which we are given a complete description of the initial state of the world and a complete description of the effects of actions, and we are asked to determine what the world will look like after a series of actions is performed. The most frequently cited example of such reasoning is probably the Yale Shooting Problem(YSP) from [HM87]. The original formalization of the problem uses the language of *situation calculus* [MH69]. (An alternative approach can be found in [KS86].) The syntax of the language contains variables of three sorts: *situation* variables $S, S', \ldots$; *fluent* variables $F, F', \ldots$;[3] and *action* variables $A, A', \ldots$[4] Its only situation constant is s_0, and $res(A, S)$ denotes the new situation that is reached after the action A is executed in situation S. The atom $holds(F, S)$ means that the fluent F is $true$ in situation S. There are also some other predicate and function symbols. The sorts of their arguments and values will be clear from their use in the rules below.

Example 2.6 In the Yale Shooting Problem(YSP), there are two fluents: $alive$ and $loaded$, and three actions: $wait$, $load$ and $shoot$. We know that the execution of loading leads to the gun being loaded, and that if the gun is shot while it is loaded, a turkey (named Fred) dies. We want to predict that after the execution of actions $load$, $wait$ and $shoot$ (in that order), Fred will be dead. It seems that the commonsense argument which leads to this conclusion

[3] A *fluent* is something that may depend on the situation, as, for instance, the location of a movable object. We will use *propositional* fluents which are assertions that can be $true$ or $false$ depending on the situation.

[4] Using a sorted language implies, first of all, that all atoms in the rules of the program are formed in accordance with the syntax of sorted predicate logic. Moreover, when we speak of an *instance* of a rule, it will be always assumed that the terms substituted for variables are of the appropriate sorts.

is based on the so-called *axiom of inertia* which says, "Things normally tend to stay the same. [MH69]" This is a typical normative statement, which in accordance with (3), can be represented by the rule:

$$y_1: \; holds(F, res(A, S)) \leftarrow holds(F, S), not\; ab(y_1, A, F, S)$$

To represent the effect of the actions *load*, *shoot* and *wait*, we need only the rule

$$y_2: \; holds(loaded, res(load, S)) \leftarrow$$

and the cancellation rule

$$y_3: \; ab(y_1, shoot, alive, S) \leftarrow holds(loaded, S)$$

which represent the priority of specific knowledge about the results of actions over the general law of inertia. Let s_0 be the initial state and suppose we are given that

$y_4: \; holds(alive, s_0)$.

Even though the resulting program $\mathcal{Y}$ consisting of y_1 to y_4 is not stratified it is possible to show (see Theorem 2.5) that it has a unique stable model. ¿From this and Lemma 2.4 it is easy to see that $\mathcal{Y}$ entails

$holds(alive, res(load, s_0))$, and

$\neg holds(alive, res(shoot, res(wait, (res(load, s_0)))))$. □

As we can see, the logic programming solution [Eva89, AB91, EK89] to the original Yale Shooting Problem is rather natural and simple. (This is not, of course, to say that it can be easily generalized to more complicated forms of reasoning about actions.) It is worth recalling that the original formulations of this story in the formalisms of circumscription and normal defaults led to unacceptable results. Some of the later solutions, in particular those from [Gel89], and [Mor88], were given in the language of autoepistemic logic and non-normal default theory respectively, and are similar to the one presented here.

Representing inheritance reasoning and reasoning about actions in logic programming is an active area of research [GL92, BG93, Dun93b, PR93, Esh88, DMB92, KS86]. Some of the works on both subjects will be discussed in the upcoming sections. We especially want to mention important challenges: formulation of more general forms of inheritance, development of theories of actions with rich ontologies and finding efficient computational means of detecting loops and dealing with floundering queries.

Existence of a unique stable model and some additional insights into the above solution can be obtained from the fact that it belongs to the class of *acyclic* programs studied in [AB91]. We will briefly describe this class and its properties.

Intuitively, the *atom dependency graph* of a program Π is analogous to the dependency graph but has as its vertices ground atoms, instead of predicate names.

Consider a program Π, whose rules with variables have been replaced by the sets of all their possible ground instantiations. The *atom dependency graph*, AD_Π, of Π consists of the ground atoms as the vertices. A triple $< P_i, P_j, s >$ is a labeled edge in AD_Π iff there is a rule r in Π with P_i in its head and P_j in its body and the label $s \in \{+, -\}$ denoting whether P_j appears in a positive or a negative literal in the body of r.

A general logic program is said to be *acyclic* if its atom dependency graph does not have a cycle.
For example, the dependency graph of a program, $\Pi = \{p(a) \leftarrow p(b)\}$ does contain a cycle with only positive edges, but the atom dependency graph of Π does not. It is also easy to see that program $\mathcal{Y}$ is acyclic.

As was shown in [AB91], most of the semantics of general logic programs coincide for this class.

The following theorem is obtained by combining results from [AB91], [Cav89], and [Prz89a] and is further discussed in Section 2.4.

Theorem 2.5 *[AB91]* Let Π be an acyclic program. Then we have:
(i) Π has a unique recursive[5] stable model;
(ii) For every ground atom A, $\Pi \models A$ iff $comp(\Pi) \cup DCA \models A$; where $comp(\Pi)$ stands for the Clark's completion of Π and DCA is the domain closure axiom [Rei80a].
(iii) For all ground atoms A that do not flounder[6] , $\Pi \models A$ iff there is an SLDNF derivation [Cla78] of A from Π. □

The first condition of the theorem guarantees that, for a rather broad class of programs (including $\mathcal{Y}$), there is an algorithm to answer all ground queries. (This is of course not true in the general case even for definite programs. As was shown in [AN78] there are definite programs with non-recursive sets of ground consequences.) The second one states that entailment in such programs Π is equivalent to classical entailment in the first-order theory $comp(\Pi) \cup DCA$. (More information on the form of this theory can be found in Section 2.4). Finally, the last condition establishes the fact that a particular, rather efficient, decision procedure, called $SLDNF$ resolution, always terminates on non-floundering ground queries of $\mathcal{Y}$. This is especially important because $SLDNF$ is incorporated in most existing Prolog interpreters.

2.3 Answering Queries

Several query answering methods have been suggested for stratified programs in the literature: in particular, SLDNF resolution [Cla78] and XOLDT resolution [TS86], [War91]. SLDNF resolution, though sound [Cla78], is only complete for a subclass of stratified programs [JLL83]. Various practical Prolog systems have been developed based on SLDNF resolution.

To answer queries with respect to programs with a multiple number of stable models, several approaches have been suggested [PAA91a, BNNS94, FLMS93, SZ90, IKH92, WC93, EK89] in the literature. Warren's XOLDT resolution uses a combination of bottom-up

[5] A set is recursive if its characteristic function is recursive.

[6] Intuitively, we say A flounders with respect to Π if while proving A from Π using SLDNF-derivation a goal is reached which contains only non-ground negative literals. For a precise definition see [Llo87].

and top-down methods. Bell et al. [BNNS94] present an approach to compute the stable models by constructing a linear programming problem from the program and solving the linear programming problem. In [Esg90], [PC89] truth maintenance systems are used to compute the stable models of a general program. Fernandez and Lobo [FL92] propose an almost top-down proof procedure to find answers to queries with respect to the stable model semantics. A "fully top-down" procedure is impossible even in propositional case since for some programs the truth of a literal w.r.t. stable model semantics can not be decided looking only at the atom dependedncy graph below it. (See for instance the Example 2.11.) As mentioned above no complete procedure (top-down or otherwise) is possible in the general case. Fernandez et al. [FLMS93] and Inoue et al. [IKH92] propose bottom-up methods to compute all the stable models of a general program. While Fernandez et al. [FLMS93] transform a general logic program to a disjunctive logic program with constraints, Inoue et al. [IKH92] transform a general program to a propositional theory. They show that the minimal (in the sense of set-theoretic inclusion) models of the resultant theory that satisfy certain conditions are the stable models of the original program.

To give the reader a flavor of the issues involved, we now present the approach of [IKH92] in more detail. Their approach is based on transforming a general logic program into a propositional theory in an extended language and reducing computing stable models of the original program to computing minimal models of the transformed theory that satisfy certain properties.

In the transformation we use new atoms that are constructed from the atoms of the original program. For each atom A, we add the new atoms A^- and A^+ to the language of the transformation. Intuitively, A^+ means A is believed to be *true* and A^- means A is not believed to be *true*.

The transformation of Π, $tr_1(\Pi)$, is obtained by translating each ground rule of the general logic program of the form (1):

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$$

to the propositional formula

$$A_1 \wedge \ldots \wedge A_m \supset (A^-_{m+1} \wedge \ldots \wedge A^-_n \wedge A_0) \vee A^+_{m+1} \vee \ldots \vee A^+_n$$

Let Π be a general logic program and $\mathcal{M}(tr_1(\Pi))$ denote the minimal models of $tr_1(\Pi)$ which satisfy the following (qualifying) properties:
(a) If a model contains A^- then it can contain neither A, nor A^+
(b) If a model contains A^+, it must also contain A.

Let $stable(\Pi) = \{S : S' \in \mathcal{M}(tr_1(\Pi))$ and S is obtained from S' by removing all atoms with $+$ and $-$ in their superscript $\}$.

Theorem 2.6 *[IKH92]* For any general logic program Π, $stable(\Pi)$ is the set of stable models of Π. □

Example 2.7 Consider the general logic program Π_1

$$p \leftarrow not\ q$$

$$q \leftarrow not\ p$$

$tr_1(\Pi_1)$ consists of the rules:

$$(q^- \wedge p) \vee q^+$$
$$(p^- \wedge q) \vee p^+$$

and has the four minimal models:
$\{q^-, p, p^-, q\}$, $\{q^-, p, p^+\}$, $\{q^+, p^-, q\}$ and $\{q^+, p^+\}$.
The first one contains p and p^- and hence is disqualified. The fourth contains p^+ and q^+ but contains neither p nor q and hence is also disqualified. The second and the third satisfy all the qualifying properties. Hence, $stable(\Pi_1)$ consists of two stable models which are obtained from the second and the third one and which are $\{p\}$ and $\{q\}$. □

There are several approaches to compute the minimal models of a positive disjunctive program [FH91, FM91]. Fernandez et al. [FLMS93] use model trees to compute minimal models. Inoue et al. [IKH92] use an extension of the model generation theorem prover (MGTP) [FH91] to directly compute the minimal models of the formulas obtained using tr_1. Obviously much more work is needed to find efficient methods of answering queries and computing the stable models of general logic programs.

2.4 Other Semantics of General Logic Programs

In this section we briefly describe some of the other approaches to the semantics of general logic programs. For more detailed discussion see the paper by Apt and Bol in this issue.[7]

The research on finding a declarative semantics for general logic programs started with the pioneering work of Clark [Cla78] and Reiter [Rei78]. Clark [Cla78] introduced the concept of *program completion* to define a declarative semantics for negation as failure. In a general logic program, the bodies of clauses with a predicate p in the head can be viewed as "sufficiency" conditions for inferring p from the program. Clark suggested that the bodies of the clauses can also be taken as "necessary" conditions, with the result that negative information about p can be assumed if all these conditions are not met. More precisely, the Clark's completion of a general logic program Π denoted by $Comp(\Pi)$ is obtained through the following steps:

Step 1: All rules in Π of the type (1) where A_0 is $p(t_1, \ldots, t_k)$, are converted to clauses of the type

$$\exists Y_1 \ldots \exists Y_s ((X_1 = t_1) \wedge \ldots \wedge (X_k = t_k) \wedge A_1 \wedge \ldots \wedge A_m \wedge \neg A_{m+1} \wedge \ldots \wedge \neg A_n) \supset p(X_1, \ldots, X_k) \quad (5)$$

where, $X_1 \ldots X_k$ are variables not appearing in the original rule, and $Y_1, \ldots, Y_s$ are variables appearing in the original rule.

[7] Since the paper of Apt and Bol in this issue is on semantics of logic programs and discusses the various semantics in detail our treatment of semantics other than the stable model semantics is brief.

Step 2: For each predicate p if

$E_1 \supset p(X_1, \dots, X_k)$
$\vdots$
$E_r \supset p(X_1, \dots, X_k)$

are all the clauses with p in its head that are generated in Step 1 (with each E_i of the form $\exists Y_1 \dots \exists Y_s((X_1 = t_1) \wedge \dots \wedge (X_1 = t_k) \wedge A_1 \wedge \dots \wedge A_m \wedge \neg A_{m+1} \wedge \dots \wedge \neg A_n)$
then $Comp(\Pi)$ contains the first order formula

$$\forall X_1 \dots \forall X_k (p(X_1, \dots, X_k) \leftrightarrow E_1 \vee \dots \vee E_r)$$

Step 3: For each predicate q if there is no rule with q in its head in the program Π then $Comp(\Pi)$ contains the first order formula

$$\forall X_1 \dots \forall X_k \neg q(X_1, \dots, X_k)$$

$Comp(\Pi)$, Clark's completion of a general logic program Π contains the first order formulas generated in Step 2 and Step 3 above and the corresponding equality theory [Cla78]. The first order formulas obtained in step 2 and step 3 above allow us to infer negative facts.

Clark's completion [Cla78] was the first declarative semantics of a general logic program. It partially corresponded to the procedural NAF rule and the SLDNF-resolution. Clark [Cla78] provided a constructive definition for *completing* a general logic program and used it to prove the soundness of the NAF rule and the SLDNF-resolution. Moreover, for a large class of programs (See for instance Theorem 2.5) the completion is computable, equivalent to the stable model semantics and sound and complete with respect to the SLDNF-resolution. The existence of Clark's declarative semantics facilitated the development of a theory of logic programs. It made possible first proofs of correctness of certain transformations of logic programs such as fold/unfold [TS84], proofs of equivalence and other properties of programs. It is still widely and successfully used for logic programming applications.

Unfortunately, Clark's semantics appears too weak for representation of some type of knowledge. Consider the following example due to Van Gelder:

Example 2.8 Suppose that we are given a graph, say,
$edge(a, b) \leftarrow$
$edge(c, d) \leftarrow$
$edge(d, c) \leftarrow$
and want to describe which vertices of the graph are reachable from a given vertex a. The following program seems to be a natural candidate for such description:
$reachable(a) \leftarrow$
$reachable(X) \leftarrow edge(Y, X), reachable(Y)$

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

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

from which such a conclusion cannot be derived. The difficulty was recognized as serious (for a good discussion of the subject see, for instance, [Prz89b]) and prompted the attempts of finding other approaches to defining the semantics of logic programs. □

¿From this point the quest for an appropriate semantics for general logic programs proceeded in several directions which can be classified broadly and incompletely into three different approaches.

The *first approach* was to put a syntactic restriction on the program. Chandra and Harel [CH85] defined the concept of stratification and Apt, Blair and Walker [ABW88] and Van Gelder [VG88] developed a fixpoint semantics for stratified programs. Przymusinski [Prz88b] generalized the concept of stratification and introduced local stratification and perfect models. The concept of local stratification was further extended by Przymusinska and Przymusinski [PP88] when they introduced the concept of weak stratification.

The *second approach* [FBJ88, Fit85, Fit86, Kun87, Kun89, LM85, Myc83, VGRS91] was to use three-valued logic instead of the classical 2-valued logic. Fitting [FBJ88, Fit85, Fit86], Kunen [Kun87, Kun89] and others [LM85, Myc83] used Kleene's strong three-valued logic while Van Gelder, Ross and Schlifp [VGRS91] used a different three valued logic to give the well-founded semantics of a logic program. Przymusinski [Prz89a, Prz89d], Dung [Dun93a], Van Gelder [VG89], and many others gave alternative formalizations of the well-founded semantics.

The semantics of Fitting and Jacob differ from the well-founded semantics. For the program consisting of the rule $p \leftarrow p$, Fitting and Jacob assign the truth value *unknown* to p while the well-founded semantics (also the perfect model semantics) assigns the value $false$ to p. The well-founded semantics is an extension of the perfect model semantics, unlike Fitting and Jacob's semantics.

The *third approach* is analogous to the traditional approach in Reiter's default logic [Rei80b] and Moore's autoepistemic logic [Moo85] in which the definition of entailment is based on the notion of beliefs. The stable model semantics [GL88] used in this paper is based on this approach. In [BS92], Baral and Subrahamanian introduce the concept of stable classes as a generalization of the stable models.

Some of these approaches aimed at preserving reduction of the notion of entailment in logic programming to entailment in "classical" two-valued or three-valued theories. Others moved closer to non-traditional non-monotonic logics.

To give the reader a flavor of these developments, we introduce the well-founded semantics, and compare it with the stable models semantics. We will follow the ideas from [BS91] and [VG89].

Definition 2.3 *[BS91]* For any general logic program Π and a set of atoms S, consider $F_\Pi(S) = a(\Pi^S)$, where a and Π^S are as in Definition 2.1 of stable models. C, a set of

interpretations[8] is said to be a *stable class* of Π iff $C = \{F_\Pi(S) : S \in C\}$.
A stable class is said to be a *strict* stable class if no proper subset of it is a stable class. $\square$

For any general logic program Π, stable models are the fixpoints of the operator F_Π. For some programs this function may not have fixpoints. The intuition behind a stable class is that even though F_Π may have no fixpoints (i.e. F_Π does not cycle around a single point), there might be a collection of interpretations so that F_Π cycles around them.

Example 2.9 Consider the following general logic program Π_3:

$a \leftarrow not\ a$

$p \leftarrow$

This program does not have any stable models. But Π_3 has two stable classes: S_0 which is the empty collection of interpretations and $S_1 = \{I_1, I_2\}$ where:

$I_1 = \{p\}$
$I_2 = \{a, p\}$

Thus, Π_3 has a unique non-empty stable class, viz. S_1, and p is $true$ in all interpretations contained in S_1. $\square$

Lemma 2.7 *[BS91, Fit91]* Let lfp stand for least fixpoint and gfp stand for greatest fixpoint. For any program Π, $\{lfp(F_\Pi^2), gfp(F_\Pi^2)\}$ is a stable class, where F_Π^2 denotes the operator that applies F_Π twice. $\square$

We now give a simple characterization of the well-founded semantics in terms of stable classes.

Definition 2.4 *[BS91]* For a general logic program Π, the stable class $\{lfp(F_\Pi^2), gfp(F_\Pi^2)\}$ defines the well-founded semantics of Π. i.e.

1. a ground atom A is $true$ in the well-founded semantics of Π iff $A \in lfp(F_\Pi^2)$, and
2. a ground atom A is $false$ in the well-founded semantics of Π iff $A \notin gfp(F_\Pi^2)$.
3. a ground atom A is $undefined$ in the well-founded semantics of Π if neither of the above two cases hold. $\square$

It was shown in [VGRS91] that unlike the stable model semantics, the well-founded semantics is defined for all general logic programs. Even though the stable model semantics is not defined for Π_3 of Example 2.9 the well-founded semantics is defined and its answer to p and a is $true$ and $unknown$ respectively.

In the following examples we show programs for which both semantics are defined but give different answers to queries.

[8] An interpretation is a set of ground atoms.

Example 2.10 Consider the following program Π_4 :

$$\left.\begin{array}{l} p \leftarrow not\, a \\ p \leftarrow not\, b \\ a \leftarrow not\, b \\ b \leftarrow not\, a \end{array}\right\} \Pi_4$$

The above program has two stable models $\{p, a\}$ and $\{p, b\}$. It has three stable classes, $\{\{p, a\}\}$, $\{\{p, b\}\}$, and $\{\{\}, \{p, a, b\}\}$. The stable class $\{\{\}, \{p, a, b\}\}$ corresponds to the well-founded semantics of the above program. Therefore, p is a consequence of Π_4 in the stable model semantics, while the answer to p in the well-founded semantics is *undefined*. □

Example 2.11 Consider the following program Π_5 [VG88]:

$$\left.\begin{array}{l} q \leftarrow not\, r \\ r \leftarrow not\, q \\ p \leftarrow not\, p \\ p \leftarrow not\, r \end{array}\right\} \Pi_5$$

Π_5 has a unique stable model, viz. $\{p, q\}$. Π_5 has three strict stable classes (stable classes which have no proper subset which is also a stable class), namely, C_1, C_2 *and* C_3, where $C_1 = \{\{q, p\}\}$, $C_2 = \{\emptyset, \{p, q, r\}\}$ and $C_3 = \{\{r\}, \{r, p\}\}$. Of these, the class C_2 corresponds to the well-founded semantics which says that p, q, r are all $undefined$. Notice that even though p is the consequence of Π_5 in the stable model semantics, its addition to Π_5 alters the set of consequences of Π_5. In particular, we will no longer be able to conclude q. □

Well-founded semantics can be considered an approximation of stable models in the sense that the well-founded semantics is correct with respect to stable model semantics [Prz90a]. By "correct" we mean that if a program has stable models, then if an atom is $true$ (resp. $false$) with respect to the well-founded semantics then it is $true$ (resp. $false$) with respect to the stable model semantics.

For the broad class of weakly stratified programs [PP90b] the well-founded semantics coincides with the stable model semantics.

There are several attempts to classify the various semantics of general logic programs. One attempt uses the notion of complexity of query answering.[9] Another attempt is based on establishing basic principles of nonmonotonic entailment. This approach was first suggested in [Gab85] and further developed by Makinson, Lehman [Leh89] and others. For a good application of this approach in the context of logic programming see [Dix91, Dix92]. Let us illustrate the main idea by introducing one such property called *cautious monotonicity* according to which an entailment relation ($\models$) defined by a logic programming semantics should satisfy the condition

$$\frac{\Pi \models a \ , \ \Pi \models b}{\Pi \cup \{a \leftarrow\} \models b} \tag{6}$$

[9] For example, in propositional case the well-founded semantics is computationally more efficient than the stable model semantics. See Section 10 for more discussion on complexity issues.

As mentioned in Example 2.11 entailment based on stable model semantics does not satisfy this property while the well-founded semantics does [Dix91]. This fact can be and often is viewed as an argument against stable model semantics. It can also be viewed as a beginning of a search for broad classes of programs for which cautious monotonicity holds. This is of course true for weakly stratified programs.

Let us suggest a broader class of programs for which we believe that the entailment defined by stable model semantics is cautiously monotonic. We say that a general logic program Π is *strongly coherent* if every subset (not necessarily proper) of Π is coherent. A similar property automatically holds for consistent first order theories. Namely, any subset of a consistent theory is consistent. This of course is not the case for logic programs (see Example 2.11 and the subset $\{p \leftarrow not\ p\}$). Hence, strong coherence may be a better analog of the consistency of first order theories than coherence. Our hypothesis is that strongly coherent programs are cautiously monotonic.

The above discussion shows that despite substantial progress the nature of the the negation as failure operator *not* is still not fully understood. Our belief is that the best way to improve the situation is to try to apply these semantics and their extensions to knowledge representation problems and to compare the elegance and efficiency of the corresponding representations. For a more comprehensive discussion on semantics of logic programs see Apt and Bol's paper in this volume.

3 Extended Logic Programs

Categorical general logic programs discussed in the previous section provide a powerful tool for knowledge representation in situations which warrant the use of the closed world assumption[10]. However, since every ground query to such programs is answered *yes* or *no*, they do not allow a programmer directly to represent incomplete knowledge about the world. To do that the language should allow for a third possibility - the *unknown* answer, which corresponds to the inability to conclude *yes* or *no*. In this section, we discuss "extended" logic programs (ELP's) [GL91] (See also [Wag91, PAA92b, PW89]) that contain a second type of negation $\neg$ (called "classical", "strong" or "explicit" by different authors who associate different meanings to it) [11] in addition to negation-as-failure *not*. General logic programs provide negative information implicitly, through closed-world reasoning; an extended logic program can include explicit negative information. In the language of extended programs, we can distinguish between a query which fails in the sense that it *does not succeed* and a query which fails in the stronger sense that its *negation succeeds*.

Formally, by an extended logic program, Π, we mean a collection of rules of the form

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n \tag{7}$$

where the L's are literals, i.e., formulas of the form p or $\neg p$, where p is an atom.

The set of all literals in the language of Π will be denoted by Lit. By $Lit(p)$ we denote the collection of ground literals formed by the predicate p. The semantics of an extended logic program assigns to it a collection of its *answer sets* - sets of literals corresponding to beliefs which can be built by a rational reasoner on the basis of Π. We will say that literal $\neg p$ is $true$ in an answer set S if $\neg p \in S$. Recall that $not\ p$ is $true$ in S if $p \notin S$. We will say that Π's answer to a literal query q is yes if q is $true$ in all answer sets of Π, no if $\bar{q}$[12] is $true$ in all answer sets of Π and *unknown* otherwise.

To give a definition of answer sets of extended logic programs, let us first consider programs without negation as failure.

The *answer set* of Π not containing *not* is the smallest (in the sense of set-theoretic inclusion) subset S of Lit such that

(i) for any rule $L_0 \leftarrow L_1, \ldots, L_m$ from Π, if $L_1, \ldots, L_m \in S$, then $L_0 \in S$;

(ii) if S contains a pair of complementary literals, then $S = Lit$.

Obviously, every program Π that does not contain negation as failure has a unique answer set which will be denoted by $b(\Pi)$.

---

[10]Informally closed world assumption or CWA [Rei78] about a statement p means that p is assumed $false$ unless there is some evidence to the contrary

[11]In this paper we refer to it as "classical" negation. As was shown by Pearce and Wagner, this negation has close ties with constructive negation of Nelson [Nel49].

[12]For any literal l, the symbol $\bar{l}$ denotes the literal opposite in sign to l. i.e. for an atom a, if $l = \neg a$ then $\bar{l} = a$, and if $l = a$ then $\bar{l} = \neg a$.

Definition 3.1 Let Π be an extended logic program without variables. For any set S of literals, let Π^S be the logic program obtained from Π by deleting

(i) each rule that has a formula *not* L in its body with $L \in S$, and

(ii) all formulas of the form *not* L in the bodies of the remaining rules.

□

Clearly, Π^S does not contain *not*, so that its answer set is already defined. If this answer set coincides with S, then we say that S is an *answer set* of Π. In other words, the answer sets of Π are characterized by the equation

$$S = b(\Pi^S). \tag{8}$$

Consider, for instance, the extended program Π_1 consisting of just one rule:

$$\neg q \leftarrow not\ p.$$

Intuitively, this rule means: "q is $false$ if there is no evidence that p is $true$." The only answer set of this program is $\{\neg q\}$. The answers that the program should give to the queries p and q are, respectively, *unknown* and *false*.

As another example, compare two programs that do not contain *not*:

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

and

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

Let's call them Π_2 and Π_3, respectively. Each of the programs has a single answer set, but these sets are different. The answer set of Π_2 is $\{\neg p\}$; the answer set of Π_3 is $\{\neg p, q\}$. Thus, our semantics is not "contrapositive" with respect to $\leftarrow$ and $\neg$; it assigns different meanings to the rules $p \leftarrow \neg q$ and $q \leftarrow \neg p$. The reason is that it interprets expressions like these as *inference rules*, rather than conditionals. (For positive programs, both points of view lead to the same semantics.) We can view this as an indication that the language of extended programs includes classical negation, but not classical implication. (From an alternative standpoint $\leftarrow$ can be viewed as a three-valued or a constructive implication and $\neg$ as some form of explicit negation.)

This approach has important computational advantages. Under rather general conditions, evaluating a query for an extended program can be reduced to evaluating two queries for a program that does not contain classical negation. Our extension of general logic programs hardly brings any new computational difficulties.

Definition 3.2 An extended logic program is said to be inconsistent if it has an inconsistent answer set. □

Proposition 3.1 An extended logic program Π is inconsistent iff Π has the unique answer set Lit. □

Syntactically, the class of general logic programs is a subclass of the class of extended logic programs. For any general logic program its stable models coincide with its answer sets. Notice however, that whenever a program not containing $\neg$ answers *no* to a query q under the stable model semantics the answer to the same query under the answer set semantics will be *unknown*.

Since general logic programs are also extended logic programs Example 2.1 is also an example of an extended logic program with no answer sets. Similarly, Example 2.2 is an example of an extended logic program with multiple answer sets.

Let us now show that extended logic programs can be reduced to general logic programs. We will need the following notation:

For any predicate p occurring in Π, let p' be a new predicate of the same arity. The atom $p'(X_1, \ldots, X_n)$ will be called the *positive form* of the negative literal $\neg p(X_1, \ldots, X_n)$. Every positive literal is, by definition, its own positive form. The positive form of a literal L will be denoted by L^+. Π^+ stands for the general logic program obtained from Π by replacing each rule (7) by

$$L_0^+ \leftarrow L_1^+, \ldots, L_m^+, not\ L_{m+1}^+, \ldots, not\ L_n^+$$

For any set $S \subset Lit$, S^+ stands for the set of the positive forms of the elements of S.

Proposition 3.2 *[GL90]* A consistent set $S \subset Lit$ is an answer set of Π if and only if S^+ is a stable model of Π^+. □

Proposition 3.2 suggests the following simple way of evaluating queries in extended logic programs. To obtain an answer for query p run queries p and p' on the program Π^+. If Π^+'s answer to p is yes then Π's answer to p is yes. If Π^+'s answer to p' is yes then Π's answer to p is no.

The next proposition is an immediate consequence of Propositions 3.2 and 2.1.

Proposition 3.3 *[GL90]* An extended logic program Π is categorical if

(a) Π^+ is stratified, and

(b) The answer set of Π^+ does not contain atoms of the form $p(t)$, $p'(t)$. □

3.1 Representing Knowledge Using Extended Logic Programs

In this section we demonstrate the applicability of extended logic programs for formalization of reasoning with incomplete information. More examples and discussions on the subject can be found in [GL91], [PAA91a], [KS90], [PAA].

Example 3.1 Let us go back to the bird's story from Example 2.5 in which we knew that birds typically fly, that penguins are exceptions to this rule - they are non-flying birds - and that our information about penguins, birds and flying objects is complete. Let us first see how this information can be expressed in the language of extended logic programs. Notice that $\mathcal{B}$ from Example 2.5, viewed as an extended logic program, fails to answer *no* to queries $penguin(tweety)$ and $flies(sam)$, and therefore is not adequate for our goal. This is, of course, not surprising, since the closed world assumption, which is responsible for the correctness of answers given by $\mathcal{B}$ under the stable model semantics, is no longer present in $\mathcal{B}$ under the answer set semantics. To represent the information correctly we need to *express the closed world assumption in the language of extended logic programs.* This can easily be done by adding to $\mathcal{B}$ the following rules

c1. $\neg bird(X) \leftarrow not\ bird(X)$

c2. $\neg penguin(X) \leftarrow not\ penguin(X)$

c3. $\neg flies(X) \leftarrow not\ flies(X)$

Notice that the program assumes that birds are the only flying objects in the universe. The resulting extended logic program $\mathcal{B}_1$ is equivalent to the original general logic program $\mathcal{B}$. □

It is possible to show that this is always the case. More precisely, we define the *closed world interpretation* $CW(\Pi)$ of a general program Π to be the extended program obtained from Π by adding the rules

$$\neg p(X_1, \ldots, X_n) \leftarrow not\ p(X_1, \ldots, X_n) \tag{9}$$

for all predicate constants p from the language of Π, where $X_1, \ldots, X_n$ are distinct variables, and n is the arity of p. The following proposition shows that the answer sets of $CW(\Pi)$ are indeed related to the answer sets of Π as we expect. Recall that HB stand for the set of all positive ground literals in the language of Π.

Proposition 3.4 If S is an answer set of a general logic program Π, then

$$S \cup \{\neg A \ :\ A \in\ HB \setminus S\} \tag{10}$$

is an answer set of $CW(\Pi)$. Moreover, every answer set of $CW(\Pi)$ can be represented in the form (10), where S is an answer set of Π.

□

Example 3.2 Let us now assume that the specification from Example 2.5 is expanded by a complete list of *wounded birds* and the following knowledge about their flying ability: wounded birds may or may not fly. Our task is to incorporate this information into the program.

Obviously, the full closed world assumption for flying birds is not applicable in this situation and should be removed from the specification. We still assume that non-birds and penguins do not fly, which can be expressed by two rules

n1. $\neg flies(X) \leftarrow penguin(X)$

n2. $\neg flies(X) \leftarrow \neg bird(X)$

Notice that the rule n2 reads as if X is not a bird then X does not fly, which corresponds to our specification. This is different from the rule

$\neg flies(X) \leftarrow not\ bird(X)$

which has an epistemic character and says that if X is not believed to be a bird then X does not fly.

The next two rules encode our general knowledge about wounded birds. The rule $i2$ cancels the application of the default 1 (see program $\mathcal{B}_2$ below) to wounded birds and, as the corresponding rule 3 for penguins, can be viewed as a formalization of the inheritance principle.

s2. $bird(X) \leftarrow wounded\_bird(X)$

i2. $ab(r1, X) \leftarrow wounded\_bird(X)$

Finally the rule $c4$ expresses the closed world assumption for wounded birds

c4. $\neg wounded\_bird(X) \leftarrow not\ wounded\_bird(X)$

Let us use these rules together with some facts about particular birds, say,

f1. $bird(tweety) \leftarrow$

f2. $penguin(sam) \leftarrow$

f3. $wounded\_bird(john) \leftarrow$

Now we will show that the program $\mathcal{B}_2$

$$\left.\begin{array}{l}
1.\ flies(X) \leftarrow bird(X), not\ ab(r1, X) \\
2.\ bird(X) \leftarrow penguin(X) \\
3.\ ab(r1, X) \leftarrow penguin(X) \\
c1.\ \neg bird(X) \leftarrow not\ bird(X) \\
c2.\ \neg penguin(X) \leftarrow not\ penguin(X) \\
c4.\ \neg wounded_bird(X) \leftarrow not\ wounded_bird(X) \\
n1.\ \neg flies(X) \leftarrow penguin(X) \\
n2.\ \neg flies(X) \leftarrow \neg bird(X) \\
s2.\ bird(X) \leftarrow wounded_bird(X) \\
i2.\ ab(r1, X) \leftarrow wounded_bird(X) \\
f1.\ bird(tweety) \leftarrow \\
f2.\ penguin(sam) \leftarrow \\
f3.\ wounded_bird(john) \leftarrow
\end{array}\right\} \mathcal{B}_2$$

has a unique consistent answer set. First let us notice that the general logic program $\mathcal{B}_2^+$ is stratified with a stratification

$P_0 = \{bird, penguin, wounded\_bird\}$

$P_1 = \{bird', penguin', wounded\_bird'\}$

$P_2 = \{ab\}$

$P_3 = \{fly',\ fly\}$

Using the Lemma 2.4, it is easy to show that there is no ground literal L such that the answer set S contains L and L^+. In virtue of Proposition 3.3, this implies that $\mathcal{B}_2$ has a unique consistent answer set. Using this fact and the Lemma 2.4, it is easy to show that $\mathcal{B}_2$'s answer to the query $flies(tweety)$ is *yes*, and that the queries $flies(sam)$ and $flies(john)$ are answered *no* and *unknown*, respectively. □

Example 3.3 Let us now modify the specification from Example 3.2 one more time by removing the closed world assumptions for all predicates from our language. Let us also assume that Tweety, Opus, and Sam are birds; Sam is a penguin; Tweety is not; and that we do not know about Opus. Notice that since Opus may be a penguin, we do not want to conclude that it flies. How can we represent this information?

The first natural idea seems to use $\mathcal{B}_2'$ obtained by removing the closed world assumptions (i.e. removing c1, c2 and c4) from $\mathcal{B}_2$. Unfortunately, this does not work. Indeed, consider the query $flies(opus)$. Since $\mathcal{B}_2'$ cannot prove that Opus is a penguin or a wounded bird, it is forced to conclude that Opus flies, which contradicts our specification.

This is again not surprising, since the corresponding cancellation axioms were written under the closed world assumptions and are too weak for an open world case. The more general form of these axioms are:

$ab(r1, X) \leftarrow not\ \neg wounded\_bird(X)$

$ab(r1, X) \leftarrow not\ \neg penguin(X)$

These axioms stop application of rule 1 for any X which *may be* a non-flying bird according to our specification. Two other necessary additions:

$\neg penguin(X) \leftarrow \neg bird(X)$

and

$\neg wounded\_bird(X) \leftarrow \neg bird(X)$

are needed to account for the contrapositive character of implication.

The resulting program $\mathcal{B}_3$

$$\left.\begin{array}{l}
1.\ flies(X) \leftarrow bird(X), not\ ab(r1, X) \\
2.\ bird(X) \leftarrow penguin(X) \\
n1.\ \neg flies(X) \leftarrow penguin(X) \\
n2.\ \neg flies(X) \leftarrow \neg bird(X) \\
s2.\ bird(X) \leftarrow wounded_bird(X) \\
f1.\ bird(tweety) \leftarrow \\
f2.\ penguin(sam) \leftarrow \\
f3.\ wounded_bird(john) \leftarrow \\
ab(r1, X) \leftarrow not\ \neg wounded_bird(X) \\
ab(r1, X) \leftarrow not\ \neg penguin(X) \\
\neg penguin(X) \leftarrow \neg bird(X) \\
\neg wounded_bird(X) \leftarrow \neg bird(X)
\end{array}\right\} \mathcal{B}_3$$

is more cautious than $\mathcal{B}_2'$. It agrees with $\mathcal{B}_2'$ on queries about Tweety and Sam but all inquires about properties of Opus (except his being a bird) are (correctly) answered as *unknown.*

The resulting program works properly if it is used in conjunction with facts formed from predicates *bird*, *penguin* and *wounded_bird*. It is also possible to show that for any query l if $\mathcal{B}_3 \models l$ then $\mathcal{B}_2 \models l$, i.e. $\mathcal{B}_3$ is correct w.r.t. $\mathcal{B}_2$.

If, however, we allow facts of the form $\neg flies(X)$, $\mathcal{B}_3$ may become inconsistent. In this case, inconsistency may be avoided by replacing the rule 1 by a weaker rule

$$flies(X) \leftarrow bird(X), not\ ab(r1, X), not\ \neg flies(X). \tag{11}$$

Let us denote the resulting program by $\mathcal{B}_4$. It is possible to show that for any set of facts not containing facts of the form $\neg flies(t)$, where t is an arbitrary ground term, programs $\mathcal{B}_3$ and $\mathcal{B}_4$ are equivalent. For a general theorem to this effect see [GP93b]. This observation leads us to a translation of normative statements to extended logic programs which is different from the one suggested in Section 2. Namely,
a normative statement of the form "A's are normally B's" is represented by the rule

$$r: \quad b(X) \leftarrow a(X), not\ ab(r, X), not\ \neg b(X) \tag{12}$$

Intuitively, the condition $not\ ab(r, X)$ in the body of (12) is used to eliminate exception to the rule r, while the condition $not\ \neg b(X)$ in the body of (12) is used to eliminate possible inconsistency because of exception to the conclusion of the rule. This more complex rule should be used only if updates of the form $\neg b(c)$ are allowed by our specification.

The weak exception to the above normative statement implying that the above default is not applicable to c's is represented by the rule

$$ab(r, X) \leftarrow not\ \neg c(X) \tag{13}$$

and the strong exception to the above normative statement implying that D's are not B's is represented by the rule

$$\neg b(X) \leftarrow d(X) \tag{14}$$

Note that the weak exceptions (wounded birds) differ from the strong exceptions (penguins). For penguins we would like to conclude that they do not fly, while for wounded birds, we do not want to conclude either that they fly, or that they do not fly. Also, we no longer need rules of the kind $ab(r, X) \leftarrow not\ \neg d(X)$. This is taken care of by the rule (12).

It should be noted that *not* is used only in particular cases: for representing normative statements and weak exceptions, for representing the CWA and for representing "unknown" information. For all other cases the classical $\neg$ is used. This is illustrated in the program $\mathcal{B}_5$ below.

Finally, let us see how we can use this program to model the behavior of the carpenter from Example 2.5. Since we are more conscious in our conclusions about flying abilities of birds, the rule 4 (from Example 2.5) becomes insufficient. It can be replaced by the following informal rule guarding the carpenter's actions: "Do not make a top of the cage for birds known to be non-flying but make it otherwise."

The following two rules formalize this in the language of extended logic programs:

$\neg make\_top(X) \leftarrow \neg flies(X)$

$make\_top(X) \leftarrow not\ \neg flies(X)$

The complete program $\mathcal{B}_5$, as given below,

$$\left.\begin{array}{l}
\neg make_top(X) \leftarrow \neg flies(X) \\
make_top(X) \leftarrow not\ \neg flies(X) \\
flies(X) \leftarrow bird(X), not\ ab(r1, X), not\ \neg flies(X) \\
2.\ bird(X) \leftarrow penguin(X) \\
n1.\ \neg flies(X) \leftarrow penguin(X) \\
n2.\ \neg flies(X) \leftarrow \neg bird(X) \\
s2.\ bird(X) \leftarrow wounded_bird(X) \\
f1.\ bird(tweety) \leftarrow \\
f2.\ penguin(sam) \leftarrow \\
f3.\ wounded_bird(john) \leftarrow \\
ab(r1, X) \leftarrow not\ \neg wounded_bird(X) \\
\neg penguin(X) \leftarrow \neg bird(X) \\
\neg wounded_bird(X) \leftarrow \neg bird(X)
\end{array}\right\} \mathcal{B}_5$$

provides an alternative model of the reasoning used by the judge to justify his decision. Because of its "cautious" approach, the authors prefer it to the formalization in Example 2.5. [BGK93] describes the precise relation between $\mathcal{B}_5$ and $\mathcal{B}$ and elaborates why $\mathcal{B}_5$ is preferable to $\mathcal{B}$.

Let us now prove that the above program, taken in conjunction with consistent (positive and negative) facts formed by predicates *bird*, *penguin*, *wounded_bird*, and *flies* is categorical.

It is easy to see that $\mathcal{B}_5^+$ is stratified with a stratification

$P_0 = \{bird, penguin, wounded\_bird\}$,

$P_1 = \{ab\}$,

$P_2 = \{flies'\}$,

$P_3 = \{flies\}$, and

$P_4 = \{make\_top, make\_top'\}$.

Now let us show that there is no constant c, such that the answer set S of $\mathcal{B}_5^+$ contains $flies(c)$ and $flies'(c)$. Suppose that $flies'(c) \in S$. By Lemma 2.4, $flies(c) \in S$ iff the premise

$bird(c), not\ ab(r1, c), not\ flies'(c)$

of rule (11) is satisfied by S, which is obviously not the case.

A similar argument works for *make_top*. By Proposition 2.2, this implies that Π is categorical. □

It is worth noting that the above techniques allow us to express priorities between defaults. Consider for instance the default "Things normally do not fly". It can be written as

$\neg flies(X) \leftarrow thing(X), not\ ab(r2, X), not\ flies(X)$

where $r2$ is the name of this rule. This default shall not be applicable to birds (which are also things), whose flying abilities are determined by more specific information. This means birds are weak exceptions to rule r2 which can be expressed by the rule

$ab(r2, X) \leftarrow not \neg bird(X)$

The resulting program together with rules expressing the subclass-class relationship between birds and things gives a correct formalization of the extended hierarchy.

The next example from [GL90] demonstrates how extended logic programs can be used to reason about unknown information in the context of deductive databases.

Example 3.4 Consider a collection of rules $\mathcal{E}_1$.

$$\left.\begin{array}{l}1.\ eligible(X) \leftarrow highGPA(X)\\ 2.\ eligible(X) \leftarrow minority(X), fairGPA(X)\\ 3.\ \neg eligible(X) \leftarrow \neg fairGPA(X), \neg highGPA(X)\\ 4.\ interview(X) \leftarrow \quad not\ eligible(X), not\ \neg eligible(X)\end{array}\right\} \mathcal{E}_1$$

used by a certain college for awarding scholarships to its students, where $highGPA$ and $fairGPA$ represent possible examination scores. The first two rules are self explanatory (we assume that variable X ranges over a given set of students). The third rule says that X is not eligible if his GPA is neither fair nor high, while the fourth rule can be viewed as a formalization of the statement:

"The students whose eligibility is not determined by the first three rules should be interviewed by the scholarship committee."

In its epistemic form the rule says : "$interview(X)$ if neither $eligible(X)$ nor $\neg eligible(X)$ is known". In general, the statement "the truth of an atomic statement p is *unknown*" can be represented by

$$not\ p, not\ \neg p. \tag{15}$$

Let us now assume that the above program is to be used in conjunction with a database DB consisting of literals specifying values of the predicates *minority*, *highGPA*, *fairGPA*. We do not assume completeness of the database. Some of the entries about the GPA and the minority status may be missing.

Consider, for instance, the DB consisting of the following two facts about one of the students:

5. $fairGPA(ann) \leftarrow$
6. $\neg highGPA(ann) \leftarrow$

(Notice that DB contains no information about the minority status of Ann.) Intuitively, it is easy to see that rules 1.–6. allow us to conclude neither $eligible(ann)$ nor $\neg eligible(ann)$. Therefore the eligibility of Ann for the scholarship is *unknown*, and by rule 4, she must be interviewed. Formally, this argument is reflected by the fact that program $\mathcal{E}_1$ consisting of rules 1.–6. has exactly one answer set:

$$\{fairGPA(ann), \neg highGPA(ann), interview(ann)\}.$$

However, if Mike is a student with highGPA or a minority student with a fairGPA the corresponding program will entail $eligible(mike)$. $\square$

The representation (15) works properly for categorical extended logic programs. The corresponding representation in the more general cases will be discussed in Example 5.3.

Example 3.5 In this example, we modify the program $\mathcal{Y}$ from Example 2.6 to allow temporal projection with incomplete knowledge about the initial situation. The law of inertia is expressed as:

$r_1: \; holds(F, res(A, S)) \leftarrow holds(F, S), not\ ab(r_1, A, F, S), not\ \neg holds(F, res(A, S))$

$r_2: \; \neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), not\ ab(r_2, A, F, S), not\ holds(F, res(A, S))$

The effects of actions are represented by

$holds(loaded, res(load, S)) \leftarrow$

and

$\neg holds(alive, res(shoot, S)) \leftarrow holds(loaded, S)$

To represent the priority of the effect rules over the inertia rule we have the cancellation rules:

$$ab(r_2, load, loaded, S) \leftarrow$$

and

$$ab(r_1, shoot, alive, S) \leftarrow not\ \neg holds(loaded, S) \tag{16}$$

Let s_0 be the initial state and suppose we are given that

$holds(alive, s_0) \leftarrow$

and

$\neg holds(loaded, s_0) \leftarrow .$

It is easy to see that the resulting program entails

$holds(alive, res(shoot, s_0))$ and

$\neg holds(alive, res(shoot, res(wait, res(load, s_0))))$.

Now suppose we have incomplete information about the initial state. i.e. we know

$holds(alive, s_0)$,

but we have no information about the gun being initially loaded. The resultant program still entails

$\neg holds(alive, res(shoot, res(load, s_0)))$

but remains undecided about

$holds(alive, res(shoot, s_0))$.

Notice that, as in the birds example above, we needed to replace the cancellation rule from Example 2.6 by a stronger rule (16). If this were not done, the program would produce the counter-intuitive conclusion
$holds(alive, res(shoot, s_0))$.

It can be shown that the above program is an extension of program $\mathcal{Y}$ and is categorical.
□

The examples demonstrate the power of extended logic programs as a knowledge representation language and outline basic ideas of the methodology of representing knowledge about action and time.

3.2 Other Semantics of Extended Logic Programs

So far we based our discussion on the answer set semantics of extended logic programs. Several other semantics of extended logic programs are suggested in the literature [AP92, PAA91c, PAA92a, PAA92b, Prz90a, KS90]. We now discuss some of them.

The formulation of well-founded semantics of general logic programs in [BS91] can be extended to define the well-founded semantics [Prz90a] of extended logic programs. More precisely, let us consider $G_\Pi(S) = b(\Pi^S)$. Then for any extended logic program Π, the fixpoints of G_Π defines the answer-set semantics, and $\{lfp(G_\Pi^2), gfp(G_\Pi^2)\}$ defines the well-founded semantics. A literal l is $true$ (resp. $false$) w.r.t. the well-founded semantics of an extended logic program Π if $l \in lfp(G_\Pi^2)$ (resp. $l \notin gfp(G_\Pi^2)$). Otherwise l is said to be $undefined$.

Pereira et al. [PAA92a] show that this definition gives unintuitive characterizations for several programs.

Example 3.6 Consider the program Π_0
$a \leftarrow not\ b$
$b \leftarrow not\ a$
$\neg a \leftarrow$
The well-founded semantics infers $\neg a$ to be $true$ and a and b to be $unknown$ with respect to the above program. Intuitively, b should be inferred $true$ and a should be inferred $false$.
□

Example 3.7 Consider the program Π_1
$b \leftarrow not\ \neg b$
and the program Π_2
$a \leftarrow not\ \neg a$
$\neg a \leftarrow not\ a$
The well-founded semantics infers b to be $true$ with respect to Π_1 and infers b to be $undefined$ with respect to $\Pi_1 \cup \Pi_2$ even though Π_2 does not have b in its language. □

To overcome the unintuitiveness of the well-founded semantics Pereira et al. [PAA92a] propose an alternative semantics of extended logic programs which we refer to as the Ω-well-founded semantics. We now define the Ω-well-founded semantics.

Definition 3.3 *[PAA92a]* Let Π be an extended logic program. $S(\Pi)$ the seminormal version of Π is obtained by replacing each rule of the form (7) by the rule:

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n, not\ \neg L_0 \tag{17}$$

$\Box$

Definition 3.4 *[PAA92a]* For any extended logic program Π, the function Ω_Π is defined as $\Omega_\Pi(X) = G_\Pi(G_{S(\Pi)}(X))$ $\Box$

Definition 3.5 *[PAA92a]* A set of literals E is said to be an Ω-extension of an extended logic program Π iff
1. E is a fixpoint of Ω_Π.
3. E is a subset of $(G_{S(\Pi)}(E))$ $\Box$

Pereira et al. [PAA92a] show that if an extended logic program has an Ω-extension then Ω_Π is a monotonic function and hence has a least fixpoint. The Ω-well-founded semantics is defined as $\{lfp(\Omega_\Pi),\ G_{S(\Pi)}(lfp(\Omega_\Pi))\}$. Entailment w.r.t. the Ω-well-founded semantics is defined as follows: A literal l is $true$ (resp. $false$) w.r.t. the Ω-well-founded semantics of an extended logic program Π if $l \in lfp(\Omega_\Pi)$ (resp. $l \notin G_{S(\Pi)}(lfp(\Omega_\Pi))$). Otherwise l is $undefined$.

Example 3.8 *[PAA92a]* Consider the following program Π_3
$c \leftarrow not\ b$
$b \leftarrow not\ a$
$a \leftarrow not\ a$
$\neg b$
The above program has $\{c, \neg b\}$ as the only Ω-extension. The Ω-well-founded semantics is given by $\{\{c, \neg b\}, \{c, a, \neg b\}\}$. $\Box$

Before we end this section we would like to briefly mention another class of semantics of extended logic programs based on contradiction removal [Dun91b, Wag93, PAA91a, GM90].

To illustrate the problem let us consider the program Π_4:
1. $p \leftarrow not\ q$
2. $\neg p \leftarrow$
3. $s \leftarrow$

Obviously, under the answer set semantics this program is inconsistent. It is possible to argue however that inconsistency of Π_4 can be localized to the rules (1.) and (2.) and should not influence the behavior of the rest of the program, i.e. Π_4's answer to query s should be *yes* and the rules causing inconsistency should be neutralized. There are several approaches to doing that. One, suggested in [KS90], modifies the answer set semantics to give preference to rules with negative conclusions (viewed as exceptions to general rules). Under the corresponding entailment relation Π_4 concludes s and $\neg p$. Another possibility is to first identify literals responsible for contradiction, in our case q. After that q can be viewed as abducible[13] and hence Π_4 will entail s, $\neg p$ and q. Another possibility arises when Ω-well-founded semantics is used as the underlying semantics of Π_4. In this case we may want to have both q and $\neg q$ undefined. This can be achieved by expanding Π_4 by new statements $q \leftarrow not\ q$ and $\neg q \leftarrow not\ \neg q$. The resulting program Π_5 entails (w.r.t. the Ω-well-founded semantics) $\neg p$ and s and infers p to be $false$. The last idea is developed to a considerable length in [PAA91a, PA93].

[13] Abducible literals are literals that can be assumed *true* if necessary. For more details see Section 8.

4 Disjunctive Logic Programs

In this section, we will discuss a further extension of the language of extended logic programs by the means necessary to represent disjunctive information about the world. Minker pioneered the use of disjunctions in the context of logic programming.[14] In [Min82] he considers positive disjunctive logic programs defined as collections of first-order clauses of the form

$$B_1 \wedge \ldots \wedge B_m \supset A_1 \vee \ldots \vee A_n \tag{18}$$

where $A's$ and $B's$ are atoms. The type of incompleteness expressible in these logic programs is, however, rather limited since their semantics suggested in [Min82] is closely related to the notion of minimal model and implicitly assumes a form of the closed world assumption. This work was generalized and/or modified by various authors (an overview can be found in [PP90a], [LMR92]) but most of the approaches still assume the closed world assumption, and hence do not allow the representation of such simple forms of incompleteness as missing information in the database tables, null values and partial definitions.

In this section, we discuss another approach to expressing disjunctive information based on the expansion of the language of extended logic programs by a new connective *or* called *epistemic disjunction* [GL91]. (Notice the use of the symbol *or* instead of classical $\vee$. The meaning of *or* is given by the semantics of disjunctive logic programs and differs from that of $\vee$. The meaning of a formula $A \vee B$ is "A is *true* or B is *true*" while a rule A *or* $B \leftarrow$ is interpreted epistemically and means "A is believed to be *true* or B is believed to be *true*." While for any atom A, $A \vee \neg A$ is always *true*, it is possible that A *or* $\neg A$ may not be *true*.)

By disjunctive logic programs we will mean a collection of rules of the form

$$L_0 \; or \ldots \; or \, L_k \leftarrow L_{k+1}, \ldots L_m, not\, L_{m+1}, \ldots, not\, L_n \tag{19}$$

where L_i's are literals. When the L_i's are atoms we refer to the program as a normal disjunctive program. When $m = n$ and the L_i's are atoms, we refer to the program as a positive disjunctive logic program.

The definition of an answer set of a disjunctive logic program Π [Prz91, GL91] is almost identical to that of extended logic programs. Let us first consider disjunctive logic programs without negation as failure.

An *answer set* of a disjunctive logic program[15] Π not containing *not* is a smallest (in a sense of set-theoretic inclusion) subset S of Lit such that

(i) for any rule $L_0 \; or \ldots \; or \, L_k \leftarrow L_{k+1} \ldots L_m$ from Π, if $L_{k+1}, \ldots, L_m \in S$, then for some i, $0 \leq i \leq k$, $L_i \in S$;

(ii) if S contains a pair of complementary literals, then $S = Lit$.

[14] Independently, Loveland [Lov87] considered extensions to Horn logic programs which he called near Horn logic programs. His main concern was efficient implementation [Lov87, SL88].

[15] For positive disjunctive logic programs this definition is similar to Minker's [Min82] original definition. For a precise relationship see Proposition 4.4.

Unlike extended logic programs without *not*, a disjunctive logic program without *not* may have more than one answer sets. For example the program

$p(a)\ or\ p(b) \leftarrow$

has two answer sets $\{p(a)\}$ and $\{p(b)\}$. We denote the answer sets of a disjunctive logic program Π that does not contain *not* by $\alpha(\Pi)$. We are now ready to define the answer set of an arbitrary disjunctive logic program.

A set S of literals is an answer set of a disjunctive logic program Π if $S \in \alpha(\Pi^S)$ where Π^S is defined in Definition 3.1.

We expand the notion of query to a formula made of literals, $\wedge$ and or. Let S be a set of literals, p be an atom and f and g be formulas.

1. p is $true$ in S if p is in S and $false$ in S if $\neg p$ is in S.
2. $f \wedge g$ is $true$ in S iff f is $true$ in S and g is $true$ in S.
3. $f \wedge g$ is $false$ in S iff f is $false$ in S or g is $false$ in S.
4. $f\ or\ g$ is $true$ in S iff f is $true$ in S or g is $true$ in S.
5. $f\ or\ g$ is $false$ in S iff f is $false$ in S and g is $false$ in S.
6. $\neg f$ is $true(false)$ in S iff f is $false$ $(true)$ in S.

A formula is said to be $true(false)$ with respect to a disjunctive logic program if it is $true(false)$ in all answer sets of the program; otherwise it is said to be *unknown*.

We again stress the difference between the epistemic or and the classical $\vee$. Consider a program consisting of the rule

$a\ or\ b \leftarrow$

This program has two answer sets $\{a\}$ and $\{b\}$. The truth of formula $(a\ or\ \neg a)$ is unknown with respect to this program, i.e., unlike $a \vee \neg a$, this formula is not a tautology.

To do some simple reasoning in disjunctive logic programs we will use a version of the "supportiveness" Lemma 2.4.

Proposition 4.1 For any answer set S of a disjunctive logic program Π:

(a) For any ground instance of a rule of the type (19) from Π, if

$\{L_{k+1} \dots L_m\} \subseteq S$ and

$\{L_{m+1} \dots L_n\} \cap S = \emptyset$

then there exists an i, $0 \leq i \leq k$ such that $L_i \in S$.

(b) If S is a consistent answer set of Π and $L \in S$ then there exists a ground instance of a rule of the type (19) from Π such that

$\{L_{k+1} \dots L_m\} \subseteq S$, and

$\{L_{m+1} \dots L_n\} \cap S = \emptyset$, and

$\{L_0 \dots L_k\} \cap S = \{L\}$. □

The definition of stratification can also be applied to disjunctive logic programs that do not contain $\neg$. The corresponding theorem guarantees existence of answer sets for such programs.

Theorem 4.2 Any stratified disjunctive logic program that does not contain $\neg$ has an answer set. $\square$

Let us look at a few simple examples of disjunctive logic programs, and their answer sets.

Let $\Pi_0 = \{p(a) \; or \; p(b) \leftarrow\}$.

It is easy to see that $\{p(a)\}$ and $\{p(b)\}$ are the only answer sets of Π_0 since they are the only minimal sets closed under its rule.

Let $\Pi_1 = \Pi_0 \cup \{r(X) \leftarrow \; not \; p(X)\}$.

Obviously, this program is stratified and hence by Theorem 4.2 has an answer set S. By part (a) of the Proposition 4.1, S must either contain $p(a)$ or contain $p(b)$. Part (b) of the Proposition 4.1 guarantees that S does not contain both. Suppose S contains $p(a)$. Then, by part (a), S contains $r(b)$, and by part (b), it contains nothing else, and hence, $\{p(a), r(b)\}$ is an answer set of Π_1. Similarly, we can show that $\{p(b), r(a)\}$ is an answer set of Π_1 and that there are no other answer sets.

4.1 Representing Knowledge Using Disjunctive Logic Programs

The following examples demonstrate the methodology of representing disjunctive information in commonsense reasoning. We start with representing the CWA in the presence of disjunctive information.

The following example shows the interplay between epistemic disjunction and the representation of the closed world assumption from previous sections.

Example 4.1 We will first use the representation of the CWA in (9). Let $\Pi_0 = \{p(a) \; or \; p(b) \leftarrow \}$, and $\Pi_2 = \Pi_0 \cup \{\neg p(X) \leftarrow \; not \; p(X)\}$ and assume that the language of Π_2 contains three constants a, b and c. It is easy to check that Π_2 has two answer sets:

$\{p(a), \neg p(b), \neg p(c)\}$ and $\{\neg p(a), p(b), \neg p(c)\}$.

and hence Π_2 answers "no" to the query $p(c)$ and answers "unknown" to the queries $p(a)$ and $p(b)$, which corresponds to our intuition.

Notice that Π_2 answers *no* to the query $p(a) \wedge p(b)$ (recall that Π_0's answer to the same query is *unknown*). This shows that the addition of the closed world assumption supplies the epistemic *or* with some degree of exclusiveness not present in it originally. The appropriateness of this effect for knowledge representation is an interesting subject for further investigation.

The effect can be avoided by using a weaker form of the CWA which views a and b as exceptions. This form can be expressed by the rules:

$\neg p(X) \leftarrow not \; p(X), not \; ab(r, X)$

$ab(r, a) \leftarrow$

$ab(r, b) \leftarrow$

where r is the name of the first rule.

Π_0 with the above three rules has the answer sets

$\{p(a), ab(r,a), ab(r,b), \neg p(c)\}$ and $\{p(b), ab(r,a), ab(r,b), \neg p(c)\}$.

Its answers to the queries $p(a)$, $p(b)$ and $p(c)$ is the same as Π_2's while its answer to the query $p(a) \wedge p(b)$ is *unknown*. □

The next example was used in [Poo89] to demonstrate difficulties with representing disjunctive information in Reiter's default logic. It is worth noting that it has a natural representation in the language of disjunctive programs.

Example 4.2 Consider the following story [Poo89]:
Normally, a person's left arm is usable, but a person with a broken left arm is an exception, and similarly for the right arm. Suppose also that we remember seeing Matt with a broken left arm or a broken right arm but we do not remember which.

Let us assume that our specification only allows updates about broken arms and that we have CWA for "broken arms" predicates.

Let us represent this information in the language of disjunctive logic programs.

Under our assumptions, the first statement of the specification can be translated into

$lh\_usable(X) \leftarrow not\ ab(l, X)$

$ab(l, X) \leftarrow lh\_broken(X)$

$rh\_usable(X) \leftarrow not\ ab(r, X)$

$ab(r, X) \leftarrow rh\_broken(X)$

The second statement may be represented as

$lh\_broken(matt)\ or\ rh\_broken(matt) \leftarrow .$

CWA about the broken arms is expressed by the following two rules.

$\neg lh\_usable(X) \leftarrow not\ lh\_usable(X)$

$\neg rh\_usable(X) \leftarrow not\ rh\_usable(X)$

The disjunctive logic program consisting of the above seven rules has two answer sets,

$\{lh\_broken(matt), ab(l, matt), rh\_usable(matt), \neg rh\_broken(matt)\}$ and

$\{rh\_broken(matt), ab(l, matt), lh\_usable(matt), \neg rh\_broken(matt)\}$ and therefore infers

$rh\_usable(matt)\ or\ lh\_usable(matt)$

which corresponds to our intended specification. Correctness of our method of representation does not depend on the above assumptions. Representations using more complicated translations of normative statements (such as 12 and 13 in Section 3.1) work equally well. However, as shown in [Poo89], similar versions of the default logic representation lead to counter-intuitive results.

□

In the next example we consider a knowledge base containing a default rule about predicate a and show that caution should be exercised when expanding the knowledge base by new disjunctive rules about a.

Example 4.3 Suppose that a language $\mathcal{L}$ contains a list of names such as $mike$, $john$, $mary$, and assume that a disjunctive logic program, Π_3, includes the following complete list of professors in a computer science department:

1. $p(mike, cs) \leftarrow$
2. $p(john, cs) \leftarrow$

To express the completeness of the list, we will again use the closed world assumption

3. $\neg p(X, Y) \leftarrow not\ p(X, Y)$.

Let us also assume that we want to represent the following information about the department:

(i) "*As a rule, professors in the computer science department have vax accounts. This rule is not applicable to Mike. He may or may not have an account.*"

In most general form this is formalized as

4. $a(X, vax) \leftarrow p(X, cs), not\ ab(r4, X), not\ \neg a(X, vax)$,

where a(X,Y) stands for "X has an account on Y," and $ab(r4, X)$ means "(4) is not applicable to X." The second statement is translated as

5. $ab(r4, mike) \leftarrow$

It is easy to see that the resulting theory entails $a(john, vax)$, but stays undecided about Mike.

Suppose now, that we have learned the following additional information:

(ii) "*every computer science professor has one of the vax or IBM accounts, but not both.*"

In the absence of any other information about computer accounts this can be represented by the rules:

6. $a(X, vax)\ or\ a(X, ibm) \leftarrow p(X, cs)$
7. $\neg a(X, ibm) \leftarrow a(X, vax), p(X, cs)$
8. $\neg a(X, vax) \leftarrow a(X, ibm), p(X, cs)$

It may appear that to find a formalization of both (i) and (ii) it suffices to merge their formalizations. Unfortunately, this does not work. To see why let us notice that we expect the resulting theory to conclude, among other things, that John has a vax account. This is, however, not the case, since the merge will have two belief sets: one containing a(john,vax) and another containing a(john,ibm). The problem occurs because of the two contrary rules (4) and (8) which can both be applied to the same professor X, and no priority is given to the rule (4). The correct solution requires a finer analysis of the situation. First we should notice that *the rule (4) should be used whenever possible* and that *the new information is only applicable to the professors which are exceptions to (4).* Two types of exceptions are possible: firstly, we may know that a professor X does not have a vax account. In this case, we should incorporate the information that X has an ibm account. This is easily done by adding the rule:

9. $a(X, ibm) \leftarrow \neg a(X, vax), p(X, cs)$.

Now the predicate a is unknown only for the professors known to be abnormal. For such professors we have reason to prefer neither *ibm* nor *vax* accounts and this lack of preference is reflected by the rule

6'. $a(X, vax)$ *or* $a(X, ibm) \leftarrow p(X, cs), ab(r4, X)$.

Our new formalization, $\mathcal{D}$ is:

$$\left.\begin{array}{l}
1.\ p(mike, cs) \leftarrow \\
2.\ p(john, cs) \leftarrow \\
3.\ \neg p(X,Y) \leftarrow not\ p(X,Y) \\
4.\ a(X, vax) \leftarrow p(X, cs), not\ ab(r4, X), not\ \neg a(X, vax) \\
5.\ ab(r4, mike) \leftarrow \\
6'.\ a(X, vax)\ or\ a(X, ibm) \leftarrow p(X, cs), ab(r4, X) \\
7.\ \neg a(X, ibm) \leftarrow p(X, cs), a(X, vax) \\
8.\ \neg a(X, vax) \leftarrow p(X, cs), a(X, ibm) \\
9.\ a(X, ibm) \leftarrow \neg a(X, vax), p(X, cs)
\end{array}\right\} \mathcal{D}$$

The new formalization, $\mathcal{D}$ implies that *john* has a *vax* account while *mike* has either a *vax* account or an *ibm* account but not both. $\mathcal{D}$ apparently satisfies the specification and may be used with any collection of facts formed by predicate symbols p and a. □

4.2 Answering Queries

There has been considerable research in developing query answering methods for positive disjunctive programs [MZ82, MR90, FM91, LMR92, HP88]. It should be noted that for positive programs minimal models coincides with answer sets. By using the renaming technique (as in Section 3) of replacing negative literals $\neg p$ by new positive atoms p' we can extend the query answering methods to answer queries in the presence of $\neg$. For disjunctive programs with *not*, Fernandez et al.and Inoue et al. [IKH92] have developed bottom-up query answering methods.

In this section we present the query answering algorithm for disjunctive logic programs of [IKH92], which is a bottom-up procedure based on computing the answer sets of a positive disjunctive logic program. It extends the approach of computing the stable models of general logic programs as described in Section 2 to compute the answer sets of disjunctive logic programs. Like the computation of the stable models, a disjunctive logic program is transformed to a disjunctive program without *not*. [IKH92] show that the answer sets of the transformed program that satisfy certain additional properties (similar to integrity constraints in databases) are the answer sets of the original disjunctive program. In the transformation we use new atoms that are constructed from the literals of the original program. For each literal L, we add the new atoms L^- and L^+ to the language of the transformation. Intuitively, L^+ means L is believed to be *true* and L^- means L is not believed to be *true*.

We also use some intermediate atoms denoted by X_i's. Recall that the symbol $\bar{L}$ denotes the literal opposite in sign to L.

The transformation of Π ([IKH92]), $tr_2(\Pi)$ is obtained by translating each rule of the disjunctive logic program of the form (19) to the following disjunctive logic program (without *not*)

$$
\begin{array}{l}
X_0 \; or \ldots or X_k \; or \; L^+_{m+1} \; or \; \ldots \; or \; L^+_n \leftarrow L_{k+1}, \ldots, L_m \\
L^-_{m+1} \leftarrow X_0 \\
\vdots \\
L^-_n \leftarrow X_0 \\
L_0 \leftarrow X_0 \\
\vdots \\
L^-_{m+1} \leftarrow X_k \\
\vdots \\
L^-_n \leftarrow X_k \\
L_k \leftarrow X_k
\end{array}
$$

Definition 4.1 Let Π be a disjunctive logic program. Let $\mathcal{M}(tr_2(\Pi))$ denote the collection of all the answer sets of $tr_2(\Pi)$ and $\mathcal{G}(tr_2(\Pi))$ denote the answer sets in $\mathcal{M}(tr_2(\Pi))$ that satisfy the following (qualifying) properties[16]:
(a)An answer set can not have both L^- and L
(b)An answer set can not have both L^- and L^+
(c)An answer set can not have both L and $\bar{L}$
(d)An answer set can not have both L^+ and $\bar{L}$
(e)An answer set can not have both L^+ and $(\bar{L})^+$
(f) If an answer set has L^+ it must also have L.

We define $answerset(\Pi)$ to be the set of minimal elements of the set $\{S : S' \in \mathcal{G}(tr_2(\Pi))$ and S is obtained from S' by removing all atoms with $+$ and $-$ in their superscript, and the intermediate atoms X_i's $\}$ $\Box$

Theorem 4.3 *[IKH92]* For any consistent disjunctive logic program Π, $answerset(\Pi)$ is the set of answer sets of Π. $\Box$

Example 4.4 *[IKH92]* Consider the following version of Example 4.2 from [IKH92].

$$
\left.
\begin{array}{l}
lh\_usable(X) \leftarrow person(X), not \; ab_1(X) \\
rh\_usable(X) \leftarrow person(X), not \; ab_2(X) \\
ab_1(X) \leftarrow \neg lh\_usable(X) \\
ab_2(X) \leftarrow \neg rh\_usable(X) \\
person(a) \leftarrow \\
\neg lh\_usable(a) \; or \; \neg rh\_usable(a) \leftarrow
\end{array}
\right\} \Pi
$$

[16]In the context of databases such properties are encoded as integrity constraints.

The transformation of Π consists of the following program

$$\left.\begin{array}{l} x_1(X)\ or\ ab_1^+(X) \leftarrow person(X) \\ ab_1^-(X) \leftarrow x_1(X) \\ lh\_usable(X) \leftarrow x_1(X) \\ x_2(X)\ or\ ab_2^+(X) \leftarrow person(X) \\ ab_2^-(X) \leftarrow x_2(X) \\ rh\_usable(X) \leftarrow x_2(X) \\ ab_1(X) \leftarrow \neg lh\_usable(X) \\ ab_2(X) \leftarrow \neg rh\_usable(X) \\ person(a) \leftarrow \\ \neg lh\_usable(a)\ or\ \neg rh\_usable(a) \leftarrow \end{array}\right\} tr_2(\Pi)$$

The answer sets of $tr_2(\Pi)$ obtained using model generation techniques are
$\{person(a), \neg lh\_usable(a), ab_1(a), rh\_usable(a), ab_1^+(a), ab_2^-(a), x_2(a)\}$
$\{person(a), \neg lh\_usable(a), ab_1(a), ab_1^+(a), ab_2^+(a)\}$
$\{person(a), \neg rh\_usable(a), ab_2(a), lh\_usable(a), ab_2^+(a), ab_1^-(a), x_1(a)\}$
$\{person(a), \neg rh\_usable(a), ab_2(a), ab_1^+(a), ab_2^+(a)\}$
The second answer set has $ab_2^+(a)$ and does not have $ab_2(a)$ and the fourth answer set has $ab_1^+(a)$ and does not have $ab_1(a)$; hence they do not satisfy property (d) of Definition 4.1. The first and the third answer set satisfy all the properties of Definition 4.1 and the answer sets of Π are obtained from them by removing all atoms with $+$ and $-$ in their superscript and the atoms with predicate x. i.e. the answer sets of Π are
$\{person(a), \neg lh\_usable(a), ab_1(a), rh\_usable(a)\}$
and $\{person(a), \neg rh\_usable(a), ab_2(a), lh\_usable(a)\}$. □

Answer sets of disjunctive logic programs can be obtained using model generation techniques [BS85, FM91, FH91, FM92, MB88]. Inoue et al. [IKH92] extend the model generation theorem prover (MGTP) [FH91] to compute the answer sets of the program obtained using tr_2. Their method avoids using the intermediate atoms X_i's.

4.3 Other Approaches to Disjunctive Logic Programs

Minker [Min82] defines the model theoretic semantics for positive disjunctive logic programs (viewed as first-order clauses of the form (18)) based on minimal Herbrand models. According to this semantics, a literal is a consequence of a disjunctive program if and only if it is *true* in every minimal Herbrand model of the program. The syntactic counterpart for inferring negative literals, called the generalized closed world assumption, is defined as follows: (We will use the terminology from [GP86].) A disjunction D of ground atoms is called *essential* w.r.t. theory T if $T \models D$ and no sub-disjunction of D is entailed by T. A ground atom is called *free for negation* in T if it does not belong to any clause essential in T. Let $\overline{T}$ be the set of negations of all ground atoms free for negation in T. Then

$GCWA(T) = \overline{T}$.

Minker [Min82] proves that for T with finite number of constants and no function symbols, $T \cup GCWA(T)$ classically entails a literal q iff q is $true$ in all minimal Herbrand models of T. This result was extended to arbitrary T in [GP86, She88].

The following proposition establishes the connection between Minker's semantics and answer set semantics of disjunctive logic programs.

Proposition 4.4 *[Gel92b]* Let Π be a program consisting of rules of the form
$A_0 \; or \ldots or \; A_k \leftarrow A_{k+1} \ldots A_m$ (where A's are atoms), and the closed world assumptions of the form
$\neg p(X) \leftarrow not \; p(X)$, for all predicates in Π.
Now let $\alpha(\Pi)$ be Π's first-order counterpart consisting of clauses
$A_{k+1} \wedge \ldots \wedge A_m \supset A_0 \vee \ldots \vee A_k$.
Then for any literal query q, Π's answer to q under answer set semantics coincides with the answer to q by $\alpha(\Pi)$ under Minker's semantics. □

Minker's GCWA was later extended by many people [YH85, GP86, GPP86] in several various directions. Several semantics have been proposed for normal disjunctive programs (disjunctive programs with *not* but without $\neg$) [LMR92, RM90, BLM92, BLM91, Ros89, Prz90b, Sak89, BED92]. The semantics proposed in [Bar92] (DWFS – Disjunctive Well-Founded Semantics) uses a fixpoint operator similar to the fixpoint operators used in [VGRS91, Prz89a] to define the well-founded semantics and iterates it starting from a "nothing is known" initial state until a fixpoint is reached. The semantics proposed in [BLM92] (GDWFS – Generalized Disjunctive Well-Founded Semantics) uses a fixpoint operator which contains an additional model theoretic part. Some of the other interesting semantics of normal disjunctive programs suggested in the literature are the extended well-founded semantics of Ross [Ros89], stationary semantics by Przymusinski [Prz90b] and the possible world semantics by Sakama [Sak89]. Minker and Ruiz [MR93] discuss extensions of the various semantics of normal disjunctive logic programs to disjunctive logic programs (They refer to it as extended disjunctive logic programs.).

The following examples give a flavor of the differences between the various semantics of disjunctive logic programs.

Example 4.5 Consider the program $\mathcal{D}_1$ consisting of the following rules:
$p(a) \; or \; p(b) \leftarrow$
$p(a) \leftarrow$
All semantics of normal disjunctive logic programs, except the possible model semantics [Sak89] and the WGCWA [RLM89, RT88] infer $p(b)$ to be $false$ with respect to the above program. The possible model semantics infers $p(b)$ to be $unknown$. □

Example 4.6 Consider the program Π_4 of Example 2.10. DWFS and stationary semantics infer p to be $unknown$ with respect to the program Π_4. But, GDWFS and the answer-set semantics infer p to be $true$. □

Example 4.7 Consider the program Π_3 of Example 2.9. DWFS and stationary semantics infer p to be $true$ with respect to the program Π_3 but Π_3 does not have any answer-sets. □

Example 4.8 Consider the following program, $\mathcal{D}_2$

$$\left.\begin{array}{l} a \leftarrow not\ b \\ a \leftarrow not\ c \\ b\ or\ c \leftarrow \end{array}\right\} \mathcal{D}_2$$

The extended well-founded semantics [Ros89], and the GDWFS do not infer a to be $true$ from the above program. But the stationary semantics, the DWFS and the answer-set semantics infer a to be $true$. $\square$

More detailed comparison of the various semantics of normal disjunctive logic programs can be found in [Bar91, Dix91, Dix92]. [Bar91] gives a framework for the various semantics based on iterating a fixpoint operator (also known as an information accumulating operator) from a "nothing is known" initial state. Dix [Dix91, Dix92] studies the various semantics based on their behavior as a nonmonotonic entailment relation.

5 Epistemic Logic Programs

The framework of disjunctive logic programs is the most general form of logic programming that we have discussed so far. In disjunctive logic programs we use two forms of negation, the classical $\neg$, and the nonmonotonic *not*. The semantics of a disjunctive logic program is defined in terms of answer sets. The answer to a query is $true$ if it is $true$ in all the answer sets. But there is no way to reason in the language itself about a particular literal being $true$ in all the (or some of the) answer sets.

The following example demonstrates the need for an extension of the language of disjunctive logic programs that will allow such reasoning:

Example 5.1 Consider the following information. We know that
(A) Either "john" or "peter" is guilty (of murder).
(B) " a person is presumed innocent if (s)he cannot be proven to be guilty"
(C) "a person can get a security clearance if we have no reason to suspect that (s)he is guilty."

Statement (A) can easily be written as a disjunctive rule:

$$A_1: \ guilty(john) \ or \ guilty(peter) \leftarrow$$

If we try to write statement (B) in the language of disjunctive logic programs using *not*, we have:

$$B_1: \ presumed\_innocent(X) \leftarrow not \ guilty(X)$$

This however is not appropriate because the program consisting of A_1 and B_1 has two answer sets $\{guilty(john), presumed\_innocent(peter)\}$ and $\{guilty(peter), presumed\_innocent(john)\}$, and therefore $presumed\_innocent(john)$ is inferred to be *unknown*. Intuitively, we should be able to infer that $presumed\_innocent(john)$ is $true$. Hence, the operator *not* in the body of B_1 is not the one we want.

Similarly, if we consider representing statement C in the language of disjunctive logic programs using *not*, we have :

$$C_1: \ cleared(X) \leftarrow not \ guilty(X)$$

But, C_1 is not appropriate, because the program consisting of A_1 and C_1 has two answer sets: $\{guilty(john), cleared(peter)\}$ and $\{guilty(peter), cleared(john)\}$, and we infer $cleared(john)$ to be *unknown*. Intuitively, we would like to infer that $cleared(john)$ is $false$.

Hence, we should expand our language and redefine answer sets in such a way that:
(B_2) We would infer $presumed\_innocent(a)$ iff there is at least one answer set that does not contain $guilty(a)$.
(C_2) We would infer $cleared(a)$ iff none of the answer sets contain $guilty(a)$. □

To capture the intuition in (B_2) and (C_2) in the above example, we use two unary operators K and M [Gel92b] and add them to our language. Intuitively, KL stands for L is *known* and ML stands for L *may be* believed. For a literal L, and a collection of sets of literals S, we say that KL is $true$ with respect to S ($S \models KL$) iff L is $true$ in *all* sets in S. ML is $true$ with respect to S ($S \models ML$) iff l is $true$ in *at least one* set in S. We say $S \models \neg KL$ iff $S \not\models KL$ and we say $S \models \neg ML$ iff $S \not\models ML$. This means $\neg KL$ is $true$ with respect to S iff there is at least one set in S where L is not $true$, and $\neg ML$ is $true$ with respect to S iff there is *no* set in S where L is $true$.

Using K and M we can represent the statements (B) and (C) in the above example by the rules:

$$innocent(X) \leftarrow \neg K guilty(X)$$

and

$$cleared(X) \leftarrow \neg M guilty(X)$$

We now define the syntax and semantics of epistemic logic programs which are obtained by adding K and M to the language of disjunctive logic programs. We refer to a literal L (without K or M) as an *objective* literal, and we refer to formulas of the form KL, ML, $\neg KL$ and $\neg ML$ as *subjective* literals.

An *epistemic logic program* is a collection of rules of the form:

$$L_1 \; or \ldots \; or \, L_k \leftarrow G_{k+1}, \ldots, G_m, not \; L_{m+1}, \ldots, not \; L_n \tag{20}$$

where the L's are objective literals and the G's are subjective or objective literals.

Let T be an epistemic logic program and S be a collection of sets of literals in the language of T. By T^S we will denote the disjunctive logic program obtained from T by:

1. removing from T all rules containing subjective literals G such that $S \not\models G$,
2. removing from rules in T all other occurrences of subjective literals.

Definition 5.1 A set S will be called a *world view* of T if S is the collection of all answer sets of T^S. Elements of S will be called *belief sets* of T. The program T^S will be called the *reduct* of T w.r.t. S. □

We now limit ourselves to epistemic programs with a unique world view.

An objective literal is said to be $true(false)$ with respect to an epistemic program if it is $true(false)$ in all elements of its world view; otherwise it is said to be unknown. A subjective literal is said to be $true(false)$ with respect to an epistemic program if it is $true(false)$ in its world view. Notice that subjective literals can not be unknown.

Example 5.2 Consider the epistemic logic program T_1:

1. $guilty(john) \; or \; guilty(peter) \leftarrow$
2. $presumed\_innocent(X) \leftarrow \neg K guilty(X)$
3. $cleared(X) \leftarrow \neg M guilty(X)$
4. $\neg presumed\_innocent(X) \leftarrow not \; presumed\_innocent(X)$

$$5.\ \neg\ cleared(X) \leftarrow not\ cleared(X)$$

Let $S_1 = \{guilty(john), presumed\_innocent(john), presumed\_innocent(peter),$
$\neg cleared(john), \neg cleared(peter)\}$

and $S_2 = \{guilty(peter), presumed\_innocent(john), presumed\_innocent(peter),$
$\neg cleared(john), \neg cleared(peter)\}$

and $S = \{S_1, S_2\}$

Since, $Mguilty(john)$ and $Mguilty(peter)$ are both $true$ with respect to S, $S \not\models \neg Mguilty(john)$ and $S \not\models \neg Mguilty(peter)$, and therefore, ${T_1}^S$ does not contain any ground instance of rule 3. Similarly, $S \models \neg Kguilty(john)$ and $S \models \neg Kguilty(peter)$ and hence ${T_1}^S$ consists of the rules:

$guilty(john)\ or guilty(peter) \leftarrow$

$presumed\_innocent(john) \leftarrow$

$presumed\_innocent(peter) \leftarrow$

$\neg presumed\_innocent(X) \leftarrow not\ presumed\_innocent(X)$

$\neg\ cleared(X) \leftarrow not\ cleared(X)$

The answer sets of ${T_1}^S$ are S_1 and S_2. Hence, S is a world view of T_1. It is possible to show that S is the only world view of T_1 [GP91] and therefore $T_1 \models presumed\_innocent(john)$, $T_1 \models presumed\_innocent(peter)$, $T_1 \models \neg cleared(john)$ and $T_1 \models \neg cleared(peter)$ which corresponds to our specification. □

Example 5.3 *[Representing Unknown]* Consider the extended logic program $\mathcal{E}_1$ from Example 3.4. Recall that it consists of rules used by a certain college for awarding scholarships to its students, and a rule saying "if the three rules do not determine the eligibility of a student then (s)he should be interviewed."

In Example 3.4 we state that for categorical extended logic programs (15) is an appropriate representation of unknown information. In this example, we show that rule (15) is not adequate for programs with multiple answer sets.

Assume that, in addition to the rules (1) - (4) of $\mathcal{E}_1$, we have the disjunction

5. $fairGPA(mike)\ or\ highGPA(mike) \leftarrow$

The epistemic logic program T_3 consisting of (1) - (4) from $\mathcal{E}_1$ and (5), has two answer sets:

$$A_1 = \{highGPA(mike), eligible(mike)\}$$

and

$$A_2 = \{fairGPA(mike), interview(mike)\},$$

and therefore the reasoner modeled by T_3 does not have enough information to establish Mike's eligibility for the scholarship (i.e. answer to $eligible(mike)$ is $unknown$). Hence, intuitively the reasoner should answer *yes* to the query $interview(mike)$. But this is not achieved by the above representation.

The intended effect is achieved by replacing (4) by the rule

$4'$. $interview(X) \leftarrow \neg K\, eligible(X),\ \ \neg K \neg eligible(X)$

The epistemic logic program, $\mathcal{E}_2$, obtained by replacing (4) in T_3 by (4') has the world view $A = \{A_1, A_2\}$ where

$$A_1 = \{highGPA(mike), eligible(mike), interview(mike)\},$$

$$A_2 = \{fairGPA(mike), interview(mike)\}.$$

Hence, $\mathcal{E}_2$ answers *unknown* to the query $eligible(mike)$ and *yes* to the query $interview(mike)$, which is the intended behavior of the system. □

Hence, in general (for theories with multiple answer sets), the statement "the truth of an atomic statement P is *unknown*" is appropriately represented by

$$not\ KP, not\ \neg KP. \tag{21}$$

So far we only considered epistemic programs with a unique world view. The following example shows epistemic programs that have multiple world views.

Example 5.4 Let T_2 consist of the rules

1. $p(a)\ or\ p(b) \leftarrow$
2. $p(c) \leftarrow$
3. $q(d) \leftarrow$
4. $\neg p(X) \leftarrow \neg M p(X)$

The specification T_2 has three world views:

$A_1 = \{\{q(d),\ p(c),\ p(a),\ \neg p(b),\ \neg p(d)\}\}$,

$A_2 = \{\{q(d),\ p(c),\ p(b),\ \neg p(a),\ \neg p(d)\}\}$, and

$A_3 = \{\ \{q(d), p(a), p(c), \neg p(d)\},\ \{q(d), p(b), p(c), \neg p(d)\}\}$.

Intuitively A_3 is preferable to the other two world views of T_2 as it treats $p(a)$ and $p(b)$ in the same manner (unlike A_1 and A_2) and can be used to answer queries with respect to T_2 For a detailed discussion on this phenomena see [Gel92b]. □

It remains to be seen if epistemic specifications with multiple world views will prove to be useful for knowledge representation.

6 Meta-logic Programming

In this section we briefly discuss another important tool for representing knowledge, in particular, knowledge in the domains containing logic programs as objects of discourse. Logic programs representing such knowledge are called meta-programs and the process of their design is referred to as meta-programming. The situations requiring various types of meta-programming are numerous. (For many interesting examples and discusion see [Kow90]). Meta-programming is used for representing and analyzing proof procedures, for assimilating new knowledge and updating knowledge bases, for modeling knowledge and beliefs of multiple agents, for hypothetical reasoning, for structured and object oriented logic programming and for many other purposes. The literature on the subject is vast. Interesting meta-programs can be found in various textbooks on logic programming. Several workshop proceedings contain papers addressing theoretical problems related to meta-programming [AR89, Met92]. Logic programming languages based on the ideas of meta-programming such as Godel [HL91], Hilog [CKW93], Reflexive Prolog [CL89] among others are beginning to gain ground in logic programming community. In this paper we will not even attempt to mention all directions of research related to meta-programming. Instead we consider a few simple (but important) meta-programs and try to outline several fundamental points related to their construction and declarative meaning.

We will start with an investigation of a two argument proof predicate

$$demo(\Pi, q)$$

(first introduced in [Kow79]), which expresses that the general logic program named Π can be used to demonstrate the conclusion named q.

The predicate *demo* and its numerous variations are used for many meta-programming applications. As an illustration let us consider its possible use for knowledge assimilation. Assume that we need to design a program representing the following relation

$$assimilate(Old, NewInfo, New)$$

between two general logic programs and an atom, where $assimilate(Old, NewInfo, New)$ means that the program New is the result of incorporating a new information $NewInfo$ into a program Old.

Let us assume that the domain of this relation consists of general logic programs and atoms written in the language $\mathcal{L}$ (called *object* programs and atoms) and that a language $\mathcal{L}_m$ (which possibly includes $\mathcal{L}$) contains some means of "naming" programs and atoms from $\mathcal{L}$. There are many different, and not equivalent ways of doing this. One of the simplest is to view rules of $\mathcal{L}$ as terms of $\mathcal{L}_m$ and to name programs by lists of their rules. For some applications such a naming scheme is both appropriate and convenient. For other applications, however, it is not. For an interesting discussion on pros and cons of different naming schemes see [Kow90].

Now we are ready to give a definition of $assimilate$. The informal specification of this relation will be developed concurrently with the formal one. The first attempt at the definition leads to a simple rule

$assimilate(Old, NewInfo, Old + NewInfo) \leftarrow$

where + is used to combine a program with an atom. We assume that $\mathcal{L}_m$ contains such an operator. Its precise implementation depends on the choice of representation of object programs and atoms in $\mathcal{L}_m$. If an object program is represented by the list of names of its rules then + can be implemented via simple *append*.

Even though in some cases this definition is sufficient it has at least two drawbacks. First, under the stable model semantics the expansion of a program $p \leftarrow not\ p, q$ by a new atom q leads to incoherency and hence some type of consistency checking is necessary. Even though the problem disappears if, say, well-founded semantics is used it reappears if classical negation is allowed in $\mathcal{L}$. For example the program
$p \leftarrow not\ q, r$
$\neg p \leftarrow$
when updated by the atom r is inconsistent under well-founded semantics. Second, indiscriminant addition of new information to a program may lead to unnecessary growth. Often, the following program, using the *demo* predicate is suggested to remedy the second problem:

$assimilate(Old, NewInfo, Old) \leftarrow demo(Old, NewInfo)$

$assimilate(Old, NewInfo, Old + NewInfo) \leftarrow not\ demo(Old, NewInfo)$

The approach works fine if our programs are monotonic, i.e. do not contain *not*. In case of nonmonotonic programs though the first rule is too strong and may lead to a loss of information. Consider for instance a program $old = \{p \leftarrow not\ q\}$ and assume that p is the new information which should be assimilated into the program *old*. Obviously, the above *assimilate* will not change the program. The problem occurs if we now learn q. The resulting program will be $new = \{p \leftarrow not\ q, \quad q \leftarrow\}$ which does not entail p, and, therefore, the observation p becomes lost.

To avoid the problem let us introduce a new predicate $monotonic(X, Y)$ (X is the monotonic part of a program Y), where the monotonic part of a program consists of the rules not containing negation as failure. Now *assimilate* can be defined as follows:

$noupdate(Old, NewInfo) \leftarrow monotonic(X, Old), demo(X, NewInfo)$

$assimilate(Old, NewInfo, Old) \leftarrow noupdate(Old, NewInfo)$

$assimilate(Old, NewInfo, Old + NewInfo) \leftarrow not\ noupdate(Old, NewInfo)$

More complicated assimilation schemes may incorporate sophisticated consistency checking and various forms of abduction but, for simplicity's sake let us assume that this scheme is the one needed for our purposes and concentrate on the definition of *demo*.

Let us first assume that the first parameter of *demo* is fixed. The following program I is commonly used to define *demo* in this case:

$demo(empty) \leftarrow$

$demo(X \& Y) \leftarrow demo(X), demo(Y)$

$demo(not(X)) \leftarrow not\ demo(X)$

$demo(X) \leftarrow clause(X, Y), demo(Y)$

For any general logic program Π, I_Π denotes the program consisting of I together with a fact of the form

$clause(A_0, A_1\& \ldots \&\ A_m\&\ not(A_{m+1})\& \ldots \&\ not(A_n)) \leftarrow$

for every rule of the form

$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$ and a fact of the form

$clause(A, empty) \leftarrow$

for every rule $A \leftarrow$ in Π.

The resulting program I_Π is called *(untyped) vanilla interpreter* for Π. Notice, that terms of the language of I_Π are build from atoms of the language of Π, constant *empty*, and two function symbols *not* and &.

In analysis of this program we will follow [MDS92a, MDS92b]. Since we use a stable model semantics we should first prove that I_Π has a stable model. Notice that I_Π is neither stratified nor locally stratified. It is possible to show however [MDS92a] that for any stratified program Π, I_Π is weakly stratified and hence categorical, i.e. has a unique stable model [PP90b]. This implies that at least for stratified programs we should not worry about existence of reasonable semantics for their vanilla meta-interpreters. So let us assume that Π is stratified. To check correctness of I_Π we need to show that for every query q, $\Pi \models q$ iff $I_\Pi \models demo(q)$. Unfortunately this is not always the case even for ground queries, i.e. I_Π is semantically incorrect. To see that consider a program Π_0 consisting of the rules:

$r(a) \leftarrow not\ q(a)$

$q(a) \leftarrow not\ p(X)$

$p(a) \leftarrow$

and assume that a is the only object constant in the language of Π_0. It is easy to see that Π_0 entails $r(a)$. This conclusion is based on the fact that the Herbrand universe of Π_0 consists of a single constant a and Π_0 is identified with the set of all of its ground instances. Now consider I_{Π_0} which consists of I and the facts

$clause(r(a), not(q(a))) \leftarrow$

$clause(q(a), not(p(X))) \leftarrow$

$clause(p(a)) \leftarrow$

Since predicate *demo* is supposed to represent entailment in Π_0 we expect I_{Π_0} to entail $demo(r(a))$. It is easy to see that this is not the case. The problem is caused by the fact that the Herbrand universe of I_{Π_0} is much richer than that of Π_0. In addition to a it contains terms $p(a), r(p(a)), \ldots$. This causes incorrect treatment of the implicit quantifiers and makes our representation of the relation *clause* incorrect. The object-level rule

$*\ q(a) \leftarrow not\ p(X)$

is implicitly universally quantified. So is its meta-level counterpart

$**\ clause(q(a), not(p(X))) \leftarrow.$

Therefore, instead of expressing that the rule (*) belongs to the program Π_0, (**) states that for any term t of the language of I_{Π_0} $q(a) \leftarrow not\ p(t)$ belongs to Π_0 which is of course not the case.

Despite this problem vanilla meta-interpreters are frequently and successfully used in practice. We discuss the explanation of this phenomena provided in [MDS92a], where the authors define a broad class of programs for which the above definition of *demo* is semantically correct. We give a slightly less general formulation here.

A rule in a program Π is called *range restricted* if any variable in the rule appears in a non-negated atom in the rule's body. A program Π is called range restricted if all its rules are range restricted.

A program $p(X) \leftarrow q(X), not\ r(X)$ is range restricted while the program Π_0 above or any program containing the rule $p(X) \leftarrow$ are not. The notion of range restriction is closely related to the notion of allowedness [TS88] and is extensively used for analysis of such notions as domain independence, floundering, and typing. It is probably fair to say that the vast majority of logic programs used in practical applications are range restricted. This observation explains the importance of the following

Theorem 6.1 *[MDS92a]* Let Π be a stratified, range restricted program. Then for every ground atom A from the language of Π,
$\Pi \models A$ iff $I_\Pi \models demo(A)$. □

The result can be generalized to the following definition of a binary predicate *demo*:

$demo(T, empty) \leftarrow$

$demo(T, X\&Y) \leftarrow demo(T, X), demo(T, Y)$

$demo(T, not(X)) \leftarrow not\ demo(T, X)$

$demo(T, X) \leftarrow clause(T, X, Y), demo(T, Y)$

where ternary *clause* is a natural generalization of its binary counterpart. This result implies correctness of the above definition of *assimilate* for stratified, range restricted programs. Most likely it can be pushed a little further. [MNR93] uses more general condition than range restriction. Stratified programs can be replaced by weakly stratified programs, or possibly, a suitable three valued version of *demo* can be designed for arbitrary range restricted programs under well-founded semantics.

Let us now look at a slightly more complicated example. Consider, for instance, the following story from [BM90].

Example 6.1 Consider a database Π_1 consisting of lists of students and courses, atomic formulas $take(s, c)$ asserting that student s has taken course c, and a definition of predicate $grad(s)$ which means that student s is eligible for graduation. We want to represent the property $near\_grad(s)$ meaning that student s is within one course of satisfying the graduation requirement.

In [BM90] this story is used to illustrate the power of intuitionistic logic programming with embedded implication $\Leftarrow$. In such a language $near\_grad$ is defined by the rule:

$near\_grad(S) \leftarrow course(C), student(S), [grad(S) \Leftarrow take(S, C)]$

Intuitively this rule states that student s is nearly a graduate if there exists some course c such that, if the student took this course, then he could graduate. Intuitionistic semantics for such programs can be found in [Mil86, BM90, GO92]. Roughly speaking, to establish the derivability of $near\_grad(s)$ in the corresponding intuitionistic program Π_1 under this semantic one should add the rule $take(s, c) \leftarrow$ to Π_1 and then try to prove $grad(s)$. As noticed in [BMPT92], similar meaning can be naturally expressed via the use of predicate $demo(T, q)$ as follows:

1. $near\_grad(S) \leftarrow course(C), student(S), demo(\Pi_1 + take(S, C), grad(S))$

To make the case slightly more interesting let us consider a program Π_2 consisting of Π_1, definitions of meta predicates *demo* and *clause*, rule (1) and the rule (2) below:

2. $eligible\_for\_scholarship(S) \leftarrow student(S), not\ near\_grad(S)$

Notice that the language of Π_2 includes both, object language of the original program and the meta language of *demo*, *clause* and other associated meta predicates. Programs with this property are called *amalgamated*. They were first introduced and investigated in [Kow79], [BK82]. To deal with such programs one should address a basic problem associated with their meaning. The problem is caused by overloading the symbols in the language. Clearly, the predicate symbols of Π_1 occur both as predicate and function symbol in Π_2. [MDS92b] argues that for a broad class of programs this does not cause a problem. Apparently, the program above belongs to this class. It will be interesting to generalize this statement to programs from [BM90] and carefully investigate the relationship between the two approaches. More complicated (but also more general) meta-programming schemes are based on naming statements of object programs by ground terms on the meta-level as in [BK82] and [HL91], on allowing sentences name themselves as in [Ric74], or by other naming devices. Normally, these approaches require the development of special semantics. Additional complications occur if we allow self-referencing meta-programs. Consider, for instance, the rule

3. $grad(X) \leftarrow started\_earlier(X, Y), near\_grad(Y), not\ ab(X)$

which says that if student X enrolled at the university a year earlier than Y and Y is near graduation, then normally X is eligible for graduation.

It seems that to incorporate this rule in Π_2 we should expand Π_2 by (3.) and replace Π_1 in rule (1.) by Π_2. This of course immediately makes the meaning of the program somewhat problematic. It is interesting to investigate to what extend such self-referencing programs are needed for knowledge representation and how their semantics are related to each other and to more conventional semantics used in the above examples. $\square$

Another intriguing question is the relationship between meta-programming and modal logics. A version of *demo* predicate formalizing entailment in classical logic was used in [KK91] to represent knowledge of multiple agents. In particular the paper contains an elegant meta-programming formalization of the three wise men problem. The object theory of the formalization is classical theory containing disjunctive information. It may be interesting to see if this development can be linked to epistemic specifications of Section 5.

7 Reasoning in Open Domains

7.1 The Semantics

In this section, we discuss the modification of the semantics of logic programs and disjunctive databases which allows for *reasoning in the absence of the domain–closure assumption* ([GP93a]). This modification increases the expressive power of the language and allows one to state explicitly the domain–closure and other assumptions about the domain of discourse in the language of logic programming.

To understand the problem, let us recall that the definition of answer set of a program was given in two steps: first the rules from Π were replaced by their ground instances, and then the definition of an answer set was given for programs not containing variables. Equating a program Π with the set of its ground instances, which occurs during the first step, was justified by the domain closure assumption [Rei80a] which asserts that *all objects in the domain of discourse have names in the language of* Π. Even though the assumption is undoubtedly useful for a broad range of applications, there are cases when it does not properly reflect the properties of the domain of discourse and causes unintended consequences. To illustrate this point, let us consider the following simple example from the literature:

Example 7.1 Consider the positive logic program Π consisting of the rule:
$p(a) \leftarrow$
and the query $q = \forall X p(X)$. □

Under the domain–closure assumption, the semantics of this program is given by its least Herbrand model [vEK76], i.e, the answer to $\forall X p(X)$ is *yes* iff for any ground term t in the language of Π the answer to $p(t)$ is *yes*. Hence, Π's answer to a query q will be *yes*. However, if we add to Π an apparently unrelated fact $r(b)$, the answer of the new program Π^* to the same query q becomes *no*. This lack of modularity, and the surprising ability of a program to entail positive facts not entailed by the corresponding classical theory were recognized as problems of the semantics of general logic programs. Przymusinski in [Prz89b] termed the above problem the *universal query problem* and suggested as a solution the semantics of general logic programs based on arbitrary (not necessarily Herbrand) minimal models. This allows him to avoid the universal query problem. Under the proper definition of the answer to a query, both Π and Π^* answer *unknown* to q. At the same time, the semantics from [Prz89b] does not diverge too far from the least Herbrand model semantics. In fact, these two semantics are equivalent for existential queries [GPP90].[17] Other solutions of the universal query problem are suggested in [Ros89], [Kun87], and [VGRS91]. They are based on the assumption that the language of any logic program contains constants and function symbols not appearing in it explicitly. Under this semantics, both programs Π and Π^* answer *no* to the query q, which, in a sense, amounts to preferring open domains over closed ones. Such a preference appears somewhat arbitrary. Unless open or closed domain assumptions are stated explicitly, *unknown* seems to be the more intuitive answer to q.

[17] T. Przymusinski's approach is not limited to positive programs. In [Prz89b], it is extended to perfect model semantics.

To obtain this answer we will use a slightly different approach. We will parametrize the definition of an answer set w.r.t. the domains of Π. For simplicity we limit ourselves to extended logic programs.

Let Π be a program over the language $\mathcal{L}_0$. To give the semantics of Π, we will first expand the alphabet of $\mathcal{L}_0$ by an infinite sequence of new constants $c_1, \ldots, c_k, \ldots$ We will call these new constants *generic*. The resulting language will be denoted by $\mathcal{L}_\infty$. By $\mathcal{L}_k$ we will denote the expansion of $\mathcal{L}_0$ by constants $c_1, \ldots, c_k$. Π_k, where $0 \leq k \leq \infty$, will stand for the set of all ground instances of Π in the language $\mathcal{L}_k$. The truth relation in the language $\mathcal{L}_k$ will be denoted by $\models_k$. The index will be omitted whenever possible.

Definition 7.1 By *k-answer set* of Π we will mean a pair $< k, B >$, where B is an answer set of Π in the language $\mathcal{L}_k$. $\square$

In the new semantics, the answer to a query q will be determined by the collection of all consistent k-answer sets which we will call *parametrized* answer sets.[18]

Example 7.2 Consider a language $\mathcal{L}_0$ over the alphabet $\{a\}$ and a general logic program Π from Example 7.1, consisting of the rule $p(a) \leftarrow$.

The following are parametrized stable models of Π:
$\{< 0, \{p(a)\} >\}$,$\{< 1, \{p(a)\} >\}$, $\{< 2, \{p(a)\} >\}, \ldots$.

$\forall X p(X)$ is *true* in the first model as the only constant in the language $\mathcal{L}_0$ is a while it is not *true* in all other models as the the corresponding languages contain constants other than a. Hence, as intended, Π's answer to the query $\forall X p(X)$ is *unknown*. $\square$

Example 7.3 Let us view rules $p(a) \leftarrow$, $q(a) \leftarrow not\ p(X)$ as an extended logic program Π over the language $\mathcal{L}_0$ with the alphabet $\{a\}$. It is easy to see that the k-answer set of Π is $< k, \{p(a)\} >$ if $k = 0$ and $< k, \{p(a), q(a)\} >$ otherwise. Hence Π's answer to the query $q(a)$ is *unknown*. $\square$

7.2 Applications

Let us start by showing that the new semantics has enough expressive power for a formalization of the domain-closure assumption.

Let Π be an arbitrary logic program in a language $\mathcal{L}_0$. We expand $\mathcal{L}_0$ by the unary predicate symbol h which stands for *named elements of the domain*. The following rules can be viewed as the definition of h:

H_1. $h(t) \leftarrow$ (for every ground term t from $\mathcal{L}_0$)

H_2. $\neg h(X) \leftarrow not\ h(X)$

The domain-closure assumption can be expressed by the rule:

DCA. $\leftarrow \neg h(X)$[19]

[18] If Π is a general logic program we will talk about parametrized stable models.

[19] A rule $\leftarrow \Gamma$ with an empty conclusion is a shorthand for the rule $\neg true \leftarrow \Gamma$. We will also assume that every logic program contains the rule $true \leftarrow$. Rules of this sort prohibit the reasoner from believing in Γ and differ from $\neg\Gamma \leftarrow$ which asserts that Γ is *false*.

Example 7.4 Let Π be the extended logic program from Example 7.3 expanded by the rules H_1, H_2 and the closed world assumptions

$\neg p(X) \leftarrow not\ p(X)$

$\neg q(X) \leftarrow not\ q(X)$

for p and q (needed for equivalence of answers obtained in general and extended logic programs). The k-answer set of Π is

$\{< 0, \{h(a), p(a), \neg q(a)\} >\}$, if $k = 0$, and

$\{< k, \{h(a), \neg h(c_1) \ldots \neg h(c_k), p(a), q(a), \neg p(c_1), \neg q(c_1) \ldots \neg p(c_k), \neg q(c_k), \} >\}$, if $k > 0$,

and therefore, Π's answer to the query $q(a)$ is *unknown*. The answer changes if Π is expanded by the domain closure assumption (DCA). The resulting program, Π_C, has the unique answer set $\{< 0, \{h(a), p(a), \neg q(a)\} >\}$ and therefore, Π_C's answer to $q(a)$ is *no*, exactly the answer produced by the answer set semantics. $\square$

It is possible to show that this is always the case, i.e. any disjunctive database Π in $\mathcal{L}_0$ under the answer set semantics is equivalent in $\mathcal{L}_0$ to the database $\Pi \cup \{H_1, H_2, DCA\}$ under the open domain semantics.

Now we will briefly discuss an example that shows the use of domain assumptions and of the concept of named objects.

Example 7.5 Consider a departmental database containing the list of courses which will be offered by a department next year, and the list of professors who will be working for the department at that time. Let us assume that the database knows the names of all the courses which may be taught by the department but, since the hiring process is not yet over, it does not know the names of all of the professors. This information can be expressed as follows:

$course(a) \leftarrow \qquad course(b) \leftarrow$

$prof(m) \leftarrow \qquad prof(n) \leftarrow$

$\neg\ course(X) \leftarrow \neg h(X)$

The k-answer set of this program is

$< k, \{course(a), course(b), \neg\ course(c_1) \ldots \neg course(c_k), prof(m), prof(n)\} >$[20]

and therefore, the above program answers *no* to the query

$\exists X\ (course(X) \wedge \neg h(X))$

and *unknown* to the query

$\exists X\ (prof(X) \wedge \neg h(X))$.

Notice that in this example, it is essential to allow for the possibility of unknown objects.

[20]Of course, the answer set also contains $h(a)$, $h(b)$, $h(m)$, $h(n)$, $\neg h(c_1) \ldots \neg h(c_k)$. In the descriptions of answer sets we will omit literals formed with h.

Let us now expand the informal specification of our database by the closed world assumptions for predicates $course$ and $prof$. The closed world assumption for $course$ says that there are no other courses except those mentioned in the database and can be formalized by the standard rule

$\neg course(X) \leftarrow not\ course(X)$.

Using this assumption, we will be able to prove that a and b are the only courses taught in our department. In the case of predicate $prof$, however, this (informal) assumption is too strong – there may, after all, be some unknown professor not mentioned in the list. However, we want to be able to allow our database to conclude that *no one known to the database is a professor unless so stated.* For that we need a weaker form of the closed world assumption, which will not be applicable to generic elements. This can easily be accomplished by the following rule:

$\neg prof(X) \leftarrow h(X), not\ prof(X)$.

The k-answer set of the resulting program Π looks as follows:

$< k, \{c(a),\ c(b),\ \neg c(m),\ \neg c(n),\ \neg c(c_1) \ldots \neg c(c_k),\ p(m),\ p(n),\ \neg p(a),\ \neg p(b)\} >$,

where c stands for $course$ and p stands for $prof$. This allows us to conclude, say, that a is not a professor without concluding that there are no professors except m and n. $\square$

Open domain semantics is also useful for representing certain types of null values in databases [GT93]. An application to formalization of anonymous exceptions to defaults [EKP91] can be found in [GP93a]. [Sch93] contains some results of query answering in open domain semantics. Related work in the context of autoepistemic logic and default logic can be found in [Lif89, Lif90]. It will be interesting to further investigate the relationship between these approaches and that of this section.

8 Abduction

Abduction is a method of reasoning which given a knowledge base T and an observation Q finds possible explanations of Q in terms of a particular set of predicates referred to as the abducible predicates. Abduction was introduced by C. Peirce in the beginning of the century (see [pei32]) and has been used in AI for explaining observations, diagnosis, planning, and natural language understanding. [kak92] gives a survey and analysis of work on the extension of logic programming to perform abductive reasoning. In this paper, we will briefly discuss the traditional role of abduction as a formalism for explanation of observations and the connection between abduction and negation as failure. We then demonstrate how abductive logic programming can be used for both deduction and explanation.

8.1 Abduction for Explaining Observations

In this subsection we briefly review the traditional view of abduction in AI as a mechanism to explain observations. C. Peirce viewed abduction as "probational adaptation of a hypothesis as explanation of observed facts, according to known rules." This intuition is approximated by the following definition:

An *abductive framework* is a triple $< T, A, I >$, where T is a knowledge base with entailment relation $\models$, I is a set of first order formulas called integrity constraints and A is a set of abducible predicates.

Let T be a theory with the first-order entailment relation. It can belong to some class of theories, e.g. Horn, or be an arbitrary collection of first-order formulas. Abductive frameworks of this sort (most frequently used in AI applications) are normally combined with definition of explanation which has the following form:

For a given set of formulas G, called observations, $\Delta \subseteq Lit(A)$ is said to be an *abductive explanation* of G with respect to $< T, A, I >$ if:

1. $T \cup \Delta \models F$ for any formula $F \in G$,
2. $T \cup \Delta$ *satisfies* I.

There are various ways to define what it means for a knowledge base KB ($T \cup \Delta$ in our case) to satisfy an integrity constraint I. The theoremhood view requires that KB satisfies I iff $KB \models I$ (i.e., KB entails every element of I). Consistency view requires $KB \cup I$ to be consistent. There are more sophisticated definitions: we will mention some of them in our further discussion.

The following example, drawn from [Pea87], can serve to illustrate a typical application of abductive frameworks to the formalization of the process of explaining observations.

Example 8.1 Consider an abductive framework $< T, A, I >$ with T (representing the background knowledge of a reasoning agent) consisting of:

$grass\_is\_wet \leftarrow rained\_last\_night$

$grass\_is\_wet \leftarrow sprinkler\_was\_on$

$shoes\_are\_wet \leftarrow grass\_is\_wet,$

the set $A = \{rained\_last\_night, sprinkler\_was\_on\}$ of abducibles and $I = \emptyset$.

The statements of T can be viewed as propositional clauses. Their intuitive meaning is self-explanatory. Suppose now, that the reasoner modeled by our abductive framework observes $G = \{shoes\_are\_wet\}$. It is easy to check that the observation G has three explanations:

$\Delta_1 = \{rained\_last\_night\}$,

$\Delta_2 = \{sprinkler\_was\_on\}$, and

$\Delta_3 = \{sprinkler\_was\_on,\ rained\_last\_night\}$,

which correspond to the commonsense explanations of G. □

The existence of multiple explanations is a general characteristic of abductive reasoning, and the selection of "preferred" explanations is an important problem. Some kind of minimality condition seems to be a natural choice. Set-theoretic minimality will leave us with the first two explanations. If later we learn that there was no rain last night and add this fact to the integrity constraints I, we will have only the second explanation left. It seems that in some commonsense arguments, more complicated preference relations on explanations are used. The discovery and investigation of such relations is an interesting topic for further research.

Let us now expand the story from the previous example by the following information: "Normally, shoes are dry". There are several possible ways to incorporate this information into the abductive framework above. To better illustrate the possible use of integrity constraints let us consider a framework $< \Pi, A, I_1 >$ where Π is a general logic program obtained by adding to T the rules

$shoes\_are\_dry \leftarrow not\ ab,$

$ab \leftarrow grass\_is\_wet,$

and $I_1 = \{\neg(shoes\_are\_dry \wedge shoes\_are\_wet)\}$.

Abductive frameworks whose first components are general logic programs are called *abductive logic programs*. Since Π is no longer a first-order theory we need a new definition of explanation. Various (non-equivalent) definitions can be found in the literature. (For comparison of several of them see [Ino92a].)

Here we shell concentrate on the proposal by Kakas and Mancarella (based on the unpublished paper by Esghi and Kowalski [EK88]). In [KM90] they develop a semantics of abductive logic programs closely related to stable model semantics of general logic program.

Let $< \Pi, A, I >$ be an abductive logic program. A set M of ground atoms is a *generalized stable model* of $< \Pi, A, I >$ if there is $\Delta \subset atoms(A)$ such that M is a stable model of $\Pi \cup \Delta$ which satisfies I. We will say that M is generated by Δ.

Δ is an *explanation* of an observation G if there is a generalized stable model M of $< \Pi, A, I >$ which is generated by Δ and satisfies all formulas from G.

It is easy to check that explanations of the observation $G = \{shoes\_are\_wet\}$ given by the program $< \Pi, A, I_1 >$ according to this definition are exactly the same as that given by the abductive framework $< T, A, I >$ above. Notice also that, due to the presence of the constraint, $< \Pi, A, I_1 >$ does not have a generalized stable model generated by $\Delta = \{\ \}$.

In addition to having a clear semantics for abduction, it is also important to have an effective method for computing abductive explanations. Theorist of [PGA87], [Poo88] provides an implementation of abduction for first-order based abductive frameworks which uses resolution-based proof procedure. There are also several procedures for answering queries for abductive logic programs and computing abductive explanations. In [EK89] a basic query answering procedure for abductive programs based on $SLDNF$ resolution is defined. In addition to the usual *yes/no* answer of $SLDNF$, this procedure also returns an abductive explanation of the corresponding query. The idea was further developed in [KM91], [Dun91a], and [DDS92]. The procedure is shown to be correct w.r.t. the stable model semantics for call-consistent logic programs, but (as pointed out in [EK89]) not in general. This fact led to modification of the procedure to achieve correctness w.r.t. the stable model semantics [SI92], as well as to work on modification of the semantics to fit the inference method of the procedure [KM91],[Dun91a]. These methods were applied to formalizations of various benchmarks in temporal, legal and other types of reasoning [Sha89, DMB92, Esh88].

There are several useful generalizations of the notion of abductive logic programs. In [Gel91], abduction is combined with reasoning with classical negation and epistemic disjunction, and generalized stable models are replaced by their answer set counterparts. In [Ino91], [Ino92a] this work is further extended by replacing abducible literals by abducible logic programs.

8.2 Abduction and Negation as Failure

There are some close similarities between abductive reasoning and negation as failure. The first attempt to make this relationship precise is due to Esghi and Kowalski. In [EK89], they give a transformation from a general logic program Π to an abductive framework $< \Pi^*, A^*, I^* >$ (where Π^* is a Horn logic program, A^* is a set of abducible predicates, and I^* is a first order theory) and show that the stable models of Π have a one-to-one correspondence with the *abductive extensions* (as defined below) of $< \Pi^*, A^*, I^* >$.

A general logic program Π is transformed to an abductive framework $< \Pi^*, A^*, I^* >$ in the following way:

- A new predicate symbol $\bar{p}$ (the opposite of p) is introduced for each p in Π, and A^* is the set of all these predicates.
- A Horn theory Π^* is obtained from Π by replacing every occurrence of literals of the form *not* $p(t)$ by $\bar{p}(t)$ (and interpreting $\leftarrow$ as an implication).
- I^* is the set of all integrity constraints of the form:
  $\forall X \neg[p(X) \wedge \bar{p}(X)]$ and
  $\forall X [p(X) \vee \bar{p}(X)]$

We say that $\Pi^* \cup \Delta$ *satisfies* integrity constraints from I if for every ground atom a from the language of Π

(a) $\Pi^* \cup \Delta \not\models a \wedge \bar{a}$ and (b) $\Pi^* \cup \Delta \models a$ or $\Pi^* \cup \Delta \models \bar{a}$

□

Proposition 8.1 *[EK89]*
1. If M is a stable model of Π then $\Pi^* \cup \{\bar{a} : a \text{ is a ground atom}, a \notin M\}$ satisfies I^*.
2. If $\Pi^* \cup \Delta$ satisfies I^* then $\{a : a \text{ is a ground atom}, \bar{a} \notin \Delta\}$ is a stable model of Π. □

The Proposition shows that an expression *not* $p(t)$ in logic programs can be interpreted as abductive hypotheses that can be assumed to hold provided that, together with the program, they satisfy a canonical set of integrity constraints.

[IS91] and [Ino91] further investigate relationship between abduction and negation as failure. In particular, they define a transformation of an abductive logic program $< \Pi, A, \emptyset >$ into an extended logic program Π^* obtained from Π by adding two rules

$p \leftarrow not\ \neg p$

$\neg p \leftarrow not\ p$

for every atom $p \in atoms(A)$.

[Ino91] shows that, under certain natural conditions on the syntax of Π, there is a simple one-to-one correspondence between generalized stable models of $< \Pi, A, \emptyset >$ and answer sets of Π^*. In [PAA91c], similar transformation under the Ω-well-founded semantics is investigated.

8.3 Combining Explanation and Deduction

In this section we introduce an entailment relation for abductive logic programs based on the notion of generalized stable model and briefly discuss its use for knowledge representation [21]. We will say that an abductive logic program T entails a formula f and write $T \models f$ if f is *true* in all generalized stable models of T. Here, unlike in extended logic programs, we are using the standard classical notion of a formula being *true* in a model. Accordingly, T answers *yes* to a query f if $T \models f$, *no* if $T \models \neg f$ and *unknown* otherwise.

Example 8.2 To illustrate the definition, let us again consider the story of birds from Example 2.5. Its formalization in abductive logic programming is given by an abductive logic program T_0, consisting of a general logic program Π_0:

1. $flies(X) \leftarrow bird(X), not\ ab(r1, X)$
2. $bird(X) \leftarrow penguin(X)$
3. $ab(r1, X) \leftarrow penguin(X)$,

the set $A = \{penguin, bird\}$ of abducibles,

and a set $I \subseteq Lit(A)$ of integrity constraints such as, say

$I = \{bird(tweety),\ \ penguin(sam)\}$.

[21] Kakas and Mancarella use generalized stable models to define explanation for observations and do not seem to have this notion of entailment mentioned explicitly. For an alternative approach based on Clark's predicate completion see, for instance, [CDT91]).

Notice, that in this formalization knowledge about particular birds is not placed in Π_0 but in the integrity constraints. The rules of Π_0 represent general knowledge about birds and their flying abilities.

Since every generalized stable model of T_0 must satisfy I, every such model contains $penguin(sam)$. Hence no generalized stable model of T_0 contains $flies(sam)$ and therefore, $T_0 \models \neg flies(sam)$. In contrast, the generalized stable model corresponding to $\Delta_0 = \{bird(tweety), penguin(sam)\}$ contains $flies(tweety)$ while the model corresponding to $\Delta_1 = \{penguin(tweety), penguin(sam)\}$ does not and therefore $T_0 \not\models flies(tweety)$. Now consider T_1 obtained from T_0 by adding to I an integrity constraint $\neg penguin(tweety)$. Obviously, $T_1 \models flies(tweety)$. □

To understand why in abductive approach certain positive facts are included in I, and not in Π it may be useful to view an abductive logic program $T =< \Pi, A, I >$ as a function $\mathcal{A}$ from 2^{Lit} to 2^{Lit} defined as

$$\mathcal{A}(X) = \{s :< \Pi, A, I \cup X > \models s\}$$

According to this view T is a program describing our general knowledge about the world while X consists of particular *observations*. Then $\mathcal{A}(X)$ is the set of facts (both positive and negative) that have to be $true$ (according to T) whenever X is $true$. This use of integrity constraints seems to be different from the original intent in which constraints were mainly used to express knowledge not easily formulated in the syntax of logic programs.

The following example may provide some additional help.

Example 8.3 Consider a simpler version of the bird example (Example 2.5) containing general rules about birds.

$$\left.\begin{array}{l} 1.\ flies(X) \leftarrow bird(X), not\ ab(r1, X) \\ 2.\ bird(X) \leftarrow penguin(X) \\ 3.\ ab(r1, X) \leftarrow penguin(X) \\ 4.\ hasbeak(X) \leftarrow bird(X) \end{array}\right\} \Pi_1$$

Let $A = \{penguin, bird\}$ and $I = \emptyset$.

Now suppose we observe $flies(a)$, i.e. $X = \{flies(a)\}$. We have two options. We can either include the observation as part of the integrity constraint or add it to the program Π_1.

If we follow the first option we can use the definition of $\mathcal{A}(X)$ and show that

$\mathcal{A}(\{flies(a)\}) = \{flies(a), bird(a), \neg penguin(a), hasbeak(a)\}$.

In the first option, the resulting abductive program $< \Pi_1, A, I \cup X >$ not only explains the observation $flies(a)$ by concluding $bird(a)$ and $\neg penguin(a)$ (which can also be achieved by the standard approach of abductive logic programs [EK88, KM90]) but also entails the conclusion $hasbeak(a)$, which is apparently not done by the standard approach.

In the second option, the resulting abductive program $< \Pi_1 \cup X, A, I >$ only entails $flies(a)$ and does not entail any of the new conclusions entailed in the previous case. □

Example 8.4 Let us now consider an extension of T_1 from Example 8.2 by including information about wounded birds as in Example 3.2 and demonstrate how this information can be represented by an abductive logic program.

Suppose that John is a wounded bird. Recall that since the degree of John's injury is unknown the corresponding abductive program should answer *unknown* to the query $flies(john)$. Obviously, postulating the exceptional status of wounded birds using a rule similar to (3.) does not produce the desired effect. We need some way of distinguishing strong and weak exceptions to defaults. A possible solution can be obtained by expanding the language of T_1 by two more abducible predicates – $badly\_wounded$ and $lightly\_wounded$ and by introducing the following rules:

5. $wounded\_bird(X) \leftarrow badly\_wounded(X)$
6. $wounded\_bird(X) \leftarrow lightly\_wounded(X)$
7. $ab(r1, X) \leftarrow badly\_wounded(X)$
8. $bird(X) \leftarrow wounded\_bird(X)$

Consider the abductive program T_2 consisting of rules 1-8 with

$I = \{wounded\_bird(john),\ \neg penguin(john),\ \ \neg(badly\_wounded \wedge lightly\_wounded)\}$ and

$A = \{penguin, bird, lightly\_wounded, badly\_wounded\}$.

To satisfy $wounded\_bird(john)$ from I the program must assume $lightly\_wounded$ or $badly\_wounded$. In the first case John will be able to fly (by rule 1) while in the second one he will not (by rule 7) and therefore T_2 answers *unknown* to the query $flies(john)$.

This methodology is, of course, rather general, and can be applied to any weak exception to an arbitrary default. □

Example 8.5 *[MD93]* In our next example, we consider an abductive logic program for the Yale shooting problem from Example 2.6. It consists of axioms (1) – (5) below

1. $holds(F, s_0) \leftarrow initially(F)$
2. $holds(F, res(A, S)) \leftarrow holds(F, S), not\ ab(A, F, S)$
3. $holds(loaded, res(load, S)) \leftarrow$
4. $ab(load, loaded, S) \leftarrow$
5. $ab(shoot, alive, S) \leftarrow holds(loaded, S)$

with $A = \{initially\}$ and the integrity constraints containing knowledge about the initial situation, say

$I = \{holds(alive, s_0), \neg holds(loaded, s_0)\}$

It is easy to see that the resulting program entails

$holds(alive, res(shoot, s_0))$ and $\neg holds(alive, res(shoot, res(load, s_0)))$.

It is possible to show that if information about the initial situation is complete, this formalization is equivalent to the one given in the language of general logic programs. But it also works well if our knowledge is incomplete. Let us assume, for instance, that we have no information about whether or not the gun is initially loaded. Assuming that all other information is unchanged, we can obtain the representation of the new situation by removing $\neg holds(loaded, s_0)$ from the old integrity constraints. The new program still entails $\neg holds(alive, res(shoot, res(load, s_0)))$ but remains undecided about $holds(alive, res(shoot, s_0))$. This is of course inexpressible in the classical paradigm. If we learned about an additional integrity constraint, say, $\neg holds(alive, res(shoot, s_0))$ then the resulting system entails $initially(loaded)$. Hence by rule 1., it also entails $holds(loaded, s_0)$ which can be viewed as an *explanation* of I. $\square$

We hope that the above discussion shows that abductive logic programming provides us with a viable alternative to more traditional extensions of logic programming. More work however is needed to better understand the role played by abduction in commonsense reasoning and the degrees in which various semantics of abductive programs reflect this role (or roles). We also need some additional insights into the mechanisms of preferring one explanations to another, into the use of integrity constraints, into computational mechanisms associated with abduction and the relationship between abductive programs and other extensions of logic programming discussed in this paper.

The notion of an abductive framework and its applications to explanations, causal reasoning, diagnoses, and other reasoning tasks was widely studied in AI ([Pea87, PGA87, CP86, Rei87b] among many others). Even though there is some obvious flow of ideas between this work and the work in abductive logic programming (for instance, the relation between abduction and negation as failure was influenced by Poole's [PGA87] *Theorist* which showed for the first time how abduction could be applied to default reasoning), much can be gained from a better understanding between the two approaches.

9 Relating logic programming and other nonmonotonic formalisms

In this section, we will briefly discuss the relationship between the logic programming–based formalisms discussed in the previous sections and various nonmonotonic logics (for a review see [Rei87a]) developed in artificial intelligence, such as circumscription [McC80], default logic [Rei80b] and autoepistemic logic [Moo85]. Even though some affinity between logic programs and nonmonotonic logics was recognized rather early [Rei82], [Lif85a], the intensive work in this direction started in 1987 after the discovery of model theoretic semantics for stratified logic programs [Apt89]. Almost immediately after this notion was introduced, stratified logic programs were mapped into the three major nonmonotonic formalisms investigated at that time: circumscription [Lif88],[Prz88a], autoepistemic logic [Gel87] and default theories [BF91], [MT89]. Research in this area was stimulated by the workshop on *Foundations of Deductive Databases and Logic Programming* [Min88] and by the workshops on *Logic Programming and Nonmonotonic Reasoning* [NMS91, PN93]. Collection of important papers can also be found in the forthcoming special issue of *Journal of Logic Programming* devoted to "logic programming and nonmonotonic reasoning". This issue includes a recent overview on the relations between logic programming and nonmonotonic reasoning [Min93] and an article on performing nonmonotonic reasoning with logic programming [PAA].

This direction of research proved to be fruitful for logic programming as well as for artificial intelligence. The results uncovered deep similarities between various, seemingly different, approaches to formalization of nonmonotonic reasoning. They helped to understand better the nature of negation as failure as a new nonmonotonic operator and led to the development of the stable model and other similar semantics of logic programs, as well as to the development of extensions of traditional logic programming with disjunction and modal operators, which apparently do not have obvious counterparts in "classical" nonmonotonic formalisms. All this greatly contributed to the better understanding and appreciation of the representational power of logic programming.

On the other hand logic programming also had a significant impact on the development of nonmonotonic logic. It not only helped to single out important classes of theories such as stratified autoepistemic theories and their variants, with comparatively good computational and other properties but also led to the development of new versions of basic formalisms, such as "default theories" [LY91, PP92], disjunctive defaults [GLPT91], reflexive autoepistemic logic [Sch91], introspective circumscription [Lif89], and MBNF [Lif91, LS90], to mention only a few. Many of these formalisms are new, and we are in the beginning stages of evaluating their utility to knowledge representation, but their role in gaining a much better understanding of commonsense reasoning cannot be seriously disputed.

In what follows, we will briefly discuss some of these results. Results relating logic programs with different semantics to various modifications of original nonmonotonic theories can be found in [PAA92a, Prz89c] among others.

9.1 Autoepistemic Logic and Logic Programming

We will start with an autoepistemic logic [Moo85] whose formulas are built from propositional atoms using propositional connectives and the modal operator B.

Definition 9.1 For any sets T and E of autoepistemic formulas, E is said to be a stable expansion of T iff
$E = Cn(T \cup \{B\phi : \phi \in E\} \cup \{\neg B\psi : \psi \notin E\})$
where Cn is a propositional consequence operator. □

Intuitively, T is a theory, the elements of T are its axioms and E is a possible model of the world together with the reasoner's beliefs. A formula F is said to be *true* in T if F belongs to all stable expansions of T. If T does not contain the modal operator B, T has a unique stable expansion [MT91]. We will denote this expansion by $Th(T)$.

Let α be a mapping [Gel87] which takes a general logic program Π, and translates its rules (of the form (7)) into autoepistemic formulas of the form
$A_1 \wedge \ldots \wedge A_m \wedge \neg B\, A_{m+1} \wedge \ldots \wedge \neg B\, A_n \supset A_0$, where B is the belief operator of autoepistemic logic. In [Gel87], it was shown that the declarative semantics of stratified logic programs can be characterized in terms of the autoepistemic theory obtained by this transformation, and that therefore, negation as failure can be understood as an epistemic operator. The stronger result establishes a one-to-one correspondence between the stable models of an arbitrary general logic program Π and the stable expansions of $\alpha(\Pi)$. Other mappings of logic programs into autoepistemic logic and its variants were investigated in [MT89], [Lif89] [Sch91], and [LS89] but none of these mappings seem to extend in a natural way to logic programs with classical negation and disjunction. Recently, several such mappings were independently found by several researchers [LS93], [MT93], [Che93]. The mapping β from [Lif93], and [Che93] translates rules of a disjunctive logic program Π (of the form (19) into autoepistemic formulas of the form
$(L_{k+1} \wedge B\, L_{k+1}) \wedge \ldots \wedge (L_m \wedge B\, L_m) \wedge \neg B\, L_{m+1} \wedge \ldots \wedge \neg B\, L_n \supset (L_0 \wedge B\, L_0) \vee (B\, L_k \wedge L_k)$.
We now state the relationship between the disjunctive logic program Π and the autoepistemic theory $\beta(\Pi)$.

Proposition 9.1 *[Lif93],[Che93]* For any disjunctive database Π, and any set A of literals in the language of Π, A is an answer set of Π iff $Th(A)$ is a stable expansion of $\beta(\Pi)$. Moreover, every stable expansion of $\beta(\Pi)$ can be represented in the above form. □

There are similar results describing mappings of disjunctive databases into reflexive autoepistemic logic [Sch91] and a logic of minimal belief and negation as failure called $MBNF$ [LW92], [LS92].

9.2 Defaults and Logic Programming

A *default* is an expression of the form

$$F \leftarrow G : \; MH_1, \ldots, MH_k, \tag{22}$$

where $F, G, H_1, \ldots, H_k$ ($k \geq 0$) are quantifier-free formulas[22] . F is the *consequent* of the default, G is its *prerequisite*, and $H_1, \ldots, H_k$ are its *justifications*. MH is interpreted as "it is consistent to believe H." A *default theory* is a set of defaults.

The operator Γ_D associated with a default theory D is defined as follows: for any set of sentences E, $\Gamma_D(E)$ is the smallest set of sentences such that

(i) for any ground instance (22) of any default from D, if $G \in \Gamma_D(E)$ and $\neg H_1, \ldots, \neg H_k \notin E$ then $F \in \Gamma_D(E)$;

(ii) $\Gamma_D(E)$ is deductively closed.

E is an *extension* for D if $\Gamma_D(E) = E$. Extensions of a default theory D play a role similar to that of stable expansions of autoepistemic theories. The simple mapping from extended logic programs to default theories identifies a rule

$$L_0 \leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n$$

with the default

$$L_0 \leftarrow L_1 \wedge \ldots \wedge \mathrm{L}_m : M\bar{L}_{m+1}, \ldots, M\bar{L}_n, \tag{23}$$

where $\bar{L}$ stands for the literal complementary to L. Every extended program is identified in this way with some default theory. It is clear that a default theory is an extended program if and only if each of its justifications and consequents is a literal, and each of its preconditions is a conjunction of literals.

Proposition 9.2 *[GL90]* For any extended program Π,

(i) if S is an answer set of Π, then the deductive closure of S is an extension of Π;

(ii) every extension of Π is the deductive closure of exactly one answer set of Π.

Thus, the deductive closure operator establishes a one–to–one correspondence between the answer sets of a program and its extensions. This result is a simple extension of results from [BF91], and [MT89]. Perhaps somewhat surprisingly, it is not easily generalized to disjunctive databases. One of the problems in finding a natural translation from such databases into default theories is related to the inability to use defaults with empty justifications in reasoning by cases: the default theory consisting of the defaults $p \leftarrow q :$, $p \leftarrow r :$ and $q \vee r \leftarrow$ does not have an extension containing p and therefore, does not entail p. This property accounts for the difficulty in formalizing Example 4.2 in default logic. It is easy to see that its disjunctive logic program counterpart entails p.

[22]We limit ourselves to the quantifier-free case. For an interesting discussion on defaults with quantifiers see [Lif90].

9.3 Truth Maintenance Systems and Logic Programming

Finally, we will briefly discuss the relationship between logic programs and nonmonotonic truth maintenance systems (TMSs) [Doy79]. Systems of this sort, originally described by procedural (and sometimes rather complicated) means, commonly serve as inference engines of AI reasoning systems. We will follow a comparatively simple description of TMSs from [Elk90]. We will need the following terminology: a *justification* is a set of directed propositional clauses of the form $\alpha \wedge \beta \supset c$ where c is an atom, α is a conjunction of atoms and β is a conjunction of negated atoms. By an interpretation we will mean a set of atoms. The justification $\alpha \wedge \beta \supset c$ *supports* the atom c w.r.t. an interpretation M if $\alpha \wedge \beta$ is *true* in M. A model M of a set of justifications Π is *grounded* if it can be written $M = \{c_1, \ldots, c_n\}$ such that each c_j has at least one justification $\alpha \wedge \beta \supset c_j$ that supports it whose positive antecedents α are a subset of $\{c_1, \ldots, c_{j-1}\}$. The task of a nonmonotonic TMS is to find a grounded model of a set of justifications Π. The form of justifications suggests the obvious analogy with rules of logic programs where negated literals $\neg A$ from β are replaced by *not* A. Let us denote the corresponding logic program by Π^*. The following theorem establishes the relationship between TMSs and logic programs:

Proposition 9.3 *[Elk90]* M is a grounded model of a collection of justifications Π iff it is a stable model of a program Π^*.

Similar results were obtained in [WB93], [GM90], [PC89], [RM89], and [FH89]. (The last two papers use autoepistemic logic instead of logic programs). They led to a better understanding of the semantics of nonmonotonic truth maintenance systems, to their use for computing stable models [Esg90] and autoepistemic extensions [JK91], for doing abductive reasoning [IS91], [RP91], and to the development of variants of TMSs based on other semantics of logic programs. A good description of one such system, based on the well-founded semantics, together with the proof of its tractability can be found in [Wit91].

10 Expressiveness and complexity results

In this section, we will briefly discuss the complexity and expressibility of logic programming languages. In [Sch92], a survey containing most of the recent results is provided.[23]

Apart from the theoretical appeal, complexity and expressiveness results for logic programming are significant for practice. Characterizations of the complexity of logic programming formalisms allow us to get insight into computational obstacles for designing efficient programs. Furthermore, by the well-developed theory of structural complexity, we often can derive immediate results on the possibility of simulating one formalism by another. Complexity is closely related to expressiveness, which, roughly speaking, measures the class of relations that a logic programming formalism can express. Expressiveness results show up limitations on the applicability of logic programming formalisms for describing particular problems.

We will assume some familiarity with complexity theory.

Let us start with the language of propositional general logic programs. The main complexity problem we will discuss is the decision problem formulated as follows: given a logic program Π and a literal l, determine whether l is a consequence of Π in the given semantics. By $|\Pi|$ and $|P|$ we will denote the number of rules and the number of propositional letters in Π respectively. Then the following holds:

Theorem 10.1 (a) the decision problem for stratified Π is $O(|\Pi|)$ (follows from [DG84]),

(b) the decision problem for Π under completion semantics and under stable model semantics is co-NP complete ([KP87] and [MT91] respectively), and

(c) the decision problem for Π under 3-valued program completion semantics and under the well-founded semantics is $O(|\Pi|)$ and $O(|\Pi|) * |P|$ respectively (folklore). $\Box$

Let us now consider programs with variables. The above results are of course only meaningful for such programs if their ground instantiations consist of a finite number of clauses. In the general case we can only talk about definability of relations defined by logic programs. Recursion-theoretic characterization of such definability provides us with insights into the expressive power of logic programming languages under different semantics.

Let us first introduce the necessary terminology:

Definition 10.1 A relation s on the set of ground terms of a language $\mathcal{L}$ is *definable* in logic programming under semantics $\models$ if there exists a program Π and predicate symbol p in the language $\mathcal{L}$ such that for every ground term t of L,
$s(t) \equiv \Pi \models p(t)$ or $s(t) \equiv \Pi \models \neg p(t)$. $\Box$

To discuss the expressive power of logic programs, we will need the following classification of formulas of second order arithmetic, i.e., arithmetic with quantifiers over sets of natural numbers [Sho67]. Variables for such sets will be denoted by X's and Y's.

[23][CS92] contains a survey on complexity results for nonmonotonic logics.

Definition 10.2 A formula is Σ_1^1 (Π_1^1) if it is of the form $\exists XF$ ($\forall XF$) where F is a first order formula. A formula is Π_2^1 if it is of the form $\forall X \exists YF$ where F is first order. $\Box$

It is well known that general Σ_1^1 formulas are far more expressive than first order formulas, and general Π_2^1 formulas are far more expressive than general Σ_1^1 formulas, and so on.

Definition 10.3 A set s of natural numbers is Σ_1^1 (Π_2^1) *definable* if it satisfies $\forall n(s(n) \equiv \Phi(n))$ where formula Φ is Σ_1^1 (Π_2^1). $\Box$

The following theorem [Sch90] [MNR93] characterizes the expressibility of the stable model semantics of logic programs over natural numbers

Theorem 10.2 *[Sch90],[MNR93]* A set s of natural numbers is Π_1^1 definable iff s is definable by a logic program under the stable model semantics. $\Box$

The actual results of [Sch90], [MNR93], and [AB90] are stronger than the above theorem. Instead of definability of sets of natural numbers, these papers deal with definability over arbitrary infinite Herbrand universes. Moreover, they show that the problem of determining if a literal l is a stable model consequence of a program Π is Π_1^1 complete, i.e. is representative of the hardest decision problem in Π_1^1. As shown in [VG89], [KP87], and [Fit85], the same result holds for the well-founded semantics as well as for two-valued and three-valued completion based semantics of logic programs.

The following results demonstrate the decrease in expressive power caused by additional restrictions on the classes of logic programs under consideration:

Theorem 10.3 *[MNR93]* A set s of natural numbers is definable by a logic program with a unique stable model iff s is Δ_1^1, i.e., s and its complement are Π_1^1 definable. $\Box$

It is known that Δ_1^1 is strictly smaller than Π_1^1, but is still highly non-recursive. The complexity is further decreased if we limit ourselves to stratified programs.

Theorem 10.4 *[AB90]* A set s of natural numbers is definable by a stratified logic program iff s is definable by a first-order arithmetic formula. $\Box$

The above theorem implies that any semantics of arbitrary logic programs is uncomputable whenever the semantics agrees with the perfect model semantics on stratified programs.

There are other interesting ways of measuring complexity of logic programs. We mention a measure, applicable to first order logic programs without function symbols, called *data complexity*. As in the theory of deductive databases, we will think of a logic program as consisting of two parts: a database of facts, plus a set of rules for inferring additional information. More precisely, we assume that the set of all predicate symbols of $\mathcal{L}$ is partitioned into the set called EDB, or extensional relations, and the set IDB, or intensional relations. The database of facts is formed from predicates in EDB and ground terms of $\mathcal{L}$, while the heads of the rules from the "rule part" are formed from predicates in IDB. Let us now fix IDB and consider the problem of checking if a ground query l is entailed by $IDB \cup D$ for a given set D of EDB facts. The corresponding complexity can be viewed as a function of the size of D. This is called *data complexity.*

Theorem 10.5 *[MT91]* The data complexity of logic programs without function symbols under the stable model semantics is co-NP complete. □

Theorem 10.6 *[VG89],[VGRS91].* The data complexity of logic programs without function symbols under the well-founded semantics is polynomial in the size of D. □

These results demonstrate that (worst-case) entailment in logic programs under the well-founded semantics is "computationally feasible", while under the stable model semantics it is not. To pay for this feasibility we lose in expressiveness. For instance, with the stable model semantics one can write a program which says that a propositional formula is satisfiable, while with the well–founded semantics one cannot.

We conclude this section by briefly addressing some recent complexity and expressiveness results for disjunctive logic programming, where the heads of clauses may be disjunctions of atoms instead of single atoms (cf. [LMR92]).

The main decision problem for the language of propositional disjunctive logic programs (not containing $\neg$) is as for nondisjunctive programs: given a disjunctive logic program Π and a literal l, determine whether l is a consequence of Π in the given semantics. Then the following holds:

Theorem 10.7 (a) the decision problem for Π under the disjunctive database rule [RT88] and the equivalent weak generalized CWA [RLM89] is polynomial [Cha93] (co-NP-complete if heads can be empty),

(b) the decision problem for Π under the possible models semantics [Sak89] and the equivalent possible worlds semantics [Cha93] is polynomial (co-NP-complete again if heads can be empty),

(c) the decision problem for Π under the careful CWA [GP86] is Π_2^P-hard and in $\Delta_3^P[O(\log n)]$, and

(d) the decision problem for Π is Π_2^P-complete for the following semantics [EG93a, EG93c, EG93d]: the generalized CWA [Min82], the extended generalized CWA [YH85], the extended CWA [GPP89], the iterated CWA [GPP89], the perfect model semantics [Prz88b], and the partial as well as total disjunctive stable model semantics [Prz91]. □

Here Π_2^P and $\Delta_3^P[O(\log n)]$ are classes above co-NP $(= \Pi_1^P)$ in the refined polynomial hierarchy, which is a subrecursive analog to the Kleene arithmetical hierarchy (cf. [Wag90]). Results for important restricted cases have been derived in [CL90]. For the extension of logic programming by classical negation [GL91], the following holds.

Theorem 10.8 The decision problem for a disjunctive logic program (containing $\neg$ and *not*) Π under the answer set semantics is Π_2^P-complete [EG93b]. □

For a special but broad class of disjunctive logic programs (in particular the headcycle-free programs) the decision problem under the answer set semantics is co-NP-complete [BED92].

Note that similar complexity results have recently been derived for various forms of nonmonotonic reasoning such as default and autoepistemic logic [Got92], nonmonotonic S4 [ST93], theory revision [EG92], and abduction [EG93e].

For programs with variables, consider the case of first order disjunctive logic programs without function symbols. Then following holds.

Theorem 10.9 *[EGM93]* (a) The data complexity of first order disjunctive logic programs without function symbols under stable model semantics is Π_2^P-complete, and

(b) The class of first order disjunctive logic programs without function symbols under stable semantics expresses Π_2^P. □

Notice that first order disjunction-free logic programs without functions expresses co-NP [Sch90]; hence, by allowing disjunction the expressive power of stable models increases a lot. For example, with disjunction we can write a program which determines whether the maximum size of a clique in a graph is odd, which is not possible by a disjunction-free program (unless the polynomial hierarchy collapses).

11 Conclusion

In this paper we considered several extensions of the language of definite logic programs and demonstrated their applicability to solving a large variety of difficult knowledge representation problems, such as formalization of normative statements, the closed world and the domain closure assumptions and their open counterparts as well as other types of default and epistemic statements. We considered several examples from such diverse domains as, reasoning in inheritance hierarchies, reasoning about result of actions, reasoning about knowledge and belief, reasoning about databases with incomplete information and several others.

Among other things we tried to demonstrate

(a) that the choice of the representation language depends on the level of completeness of knowledge about the problem. For instance, when knowledge is assumed complete the language of general logic program seems to be appropriate. When the only form of incompleteness is complete lack of knowledge (like missing entries in database tables) the language of extended logic programs seems to be the language of choice. Representation of various forms of partial knowledge requires disjunctive logic programs and their extensions or abductive frameworks.

(b) that the choice of particular representation of normative statements and other constructs of natural language depends not only on the representation language but also on the type of updates that are allowed. For example, if the only updates allowed by the specification are of the form $q(t)$ the normative statement "p's are normally q's" is translated as
$p(X) \leftarrow q(X), not\ ab(X)$

If we allow updates consisting of literals formed with predicate p then the above normative statement should be translated as
$p(X) \leftarrow q(X), not\ ab(X), not\ \neg p(X)$

Fortunately, in many cases the more complex formalization seem to be correct extensions of the simpler ones [BGK93, Tur93]. The precise relationship between these different approaches is an interesting subject for further investigation.

We believe that the proposed formalizations compare favorably with those given in traditional nonmonotonic formalisms: recall for instance that Example 2.6 poses a serious problem for Circumscription and Example 4.2 causes difficulties for Reiter's Default logic. It is not clear how to represent epistemic reasoning, say from Example 5.1, in any known form of nonmonotonic modal logic. However, extended logic programming languages allow all these examples to be treated in a uniform fashion.

Restrictive syntax of logic programming languages facilitates the adoption of query answering methods developed in the context of traditional logic programming and deductive databases to more complicated forms of knowledge representation and reasoning. It also helps to avoid another problem associated with the use of superclassical logics: existence of several natural, but nonequivalent translations from natural language statements into the formalism. Consider for instance Example 4.2. Simple disjunctive statement "Matt's left or right hand is broken" can be translated in, say, MBNF [LW92], as

$lh\_broken \lor rh\_broken$

or as

$B\ lh\_broken \lor B\ rh\_broken$

where B is the belief operator of MBNF. Only the second translation (probably the less obvious one) leads to the correct result.

There are of course many remaining problems. Even though in many cases application of our techniques led to modular representation of knowledge [24] a greater degree of modularity is desirable. It remains to be seen if this can only be achieved by limiting the types of updates allowed by our formalization. It is well known in the theory of data structures that data representation is dependent on the operations allowed on the data. Realization of its importance led to the developments of abstract data types. Similar considerations can lead to the discovery of interesting knowledge structures. It is also possible that more work on the methodology of representing knowledge and/or extension of the language can help to solve the problem of modular updates. The problem of updates is closely related to the more general problem of belief revision. A better understanding of this process and development of its mathematical models in the context of logic programming paradigms presents an interesting and important challenge to the logic programming community.

Another important question which requires much more work is the development of query-answering systems for the new languages. Even though traditional inference methods of Prolog and deductive databases are easily adaptable to categorical extended logic programs much improvement is needed in developing methods computing well-founded, stable and/or

[24] Formalization of knowledge is called modular if small changes in the informal knowledge base cause small changes in its formal counterpart.

other types of semantics. Very promising work in this direction is done by Pereira et al. [PAA91b], Chen and Warren [WC93] and others. Some of the other important issues are to learn to deal with floundering queries and planning problems, and to incorporate new logic programming paradigms such as constrained logic programming and concurrent logic programming in the nonmonotonic framework. Extensions of the languages by allowing more complex data such as sets and aggregates are very important in database applications [KS91, BRSS92, Gel92a].

Even more questions remain for noncategorical programs, i.e. logic programs with multiple answer sets or disjunctive logic programs. In this case one of the most important problems seems to be the lack of clear procedural interpretation of rules of a program. Such interpretation of definite logic programs which treats predicates as procedure calls and interpreters the rule $A_0 \leftarrow A_1, \ldots A_n$ as saying "to execute procedure A_0 execute procedures $A_1, \ldots A_n$" was suggested by Kowalski in [Kow74] and lies at the heart of logic programming paradigm. It remains to be seen if similar interpretation can be discovered for disjunctive logic programs and the other formalisms described in the paper.

Another crucial direction of research is related to using extensions of logic programming to representing knowledge in particular domains. Theories of actions and time, representing null values and other forms of incomplete information in databases, legal reasoning and reasoning about diagnoses is an incomplete list of interesting examples. Building theories is a slow process which only succeeds if new ideas are built on the old ones. To learn how to do that in a careful mathematical way is one of the major challenges faced by the field. Finally, we would like to mention that real progress in all of these areas is tightly related to building mathematical theory of declarative logic programming. We hope that the paper showed that such a theory, even though it is still in the stage of infancy, contains some nontrivial results and methods. We also hope that the paper will contribute to its further development.

Acknowledgments

A lot of people (knowingly or unknowingly) contributed to this survey by sharing with us their views on the field and particular results discussed in the paper. To all of them - our deepest thanks. We would like to thank all five referees whose suggestions and criticism help us to improve the paper. Special thanks also to G. Gottlob for his help on the complexity section. This work was supported in part by the grants NSF-IRI-92-11-662, NSF-CDA-90-15-006, and NSF-IRI-91-03-112.

References

[AB90] K. Apt and H. Blair. Arithmetic classification of perfect models of stratified programs. *Fundamenta Informaticae*, 13:1–18, 1990.

[AB91] K. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3,4):335–365, 1991.

[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 89–148. Morgan Kaufmann, San Mateo, CA., 1988.

[AN78] H. Andreka and I. Nemeti. The generalized completeness of Horn predicate logic as a programming language. *Acta Cybernetica*, 4:3–10, 1978.

[AP92] J. Alferes and L. Pereira. On logic program semantics with two kinds of negation. In K. Apt, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming, Wash DC*, pages 574–588. MIT Press, Nov 1992.

[Apt89] K.R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North Holland, 1989.

[AR89] H. Abramson and M. Rogers, editors. *Meta-Programming in Logic Programming*. MIT Press, 1989.

[Bar91] C. Baral. *Issues in Knowledge Representation : Semantics and Knowledge Combination*. PhD thesis, University of Maryland (CS-TR-2761), Department of Computer Science, August 1991.

[Bar92] C. Baral. Generalized Negation As Failure and Semantics of Normal Disjunctive Logic Programs. In A Voronkov, editor, *Proceedings of International Conference on Logic Programming and Automated Reasoning, St. Petersburg*, pages 309–319, 1992.

[BED92] R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. In *Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming*, pages 813–827, 1992.

[BF91] N. Bidoit and C. Froidevaux. General logical databases and programs: Default logic semantics and stratification. *Journal of Information and Computation*, 91(1):15–54, 1991.

[BG93] C. Baral and M. Gelfond. Representing concurrent actions in extended logic programming. In *Proc. of 13th International Joint Conference on Artificial Intelligence, Chambery, France*, pages 866–871, 1993.

[BGK93] C. Baral, M. Gelfond, and O. Kosheleva. Approximating general logic programs, 1993. In *Proc. of the International Symposium on Logic Programming*, pages 181–198, 1993.

[BK82] K. Bowen and R. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, pages 153–173. Academic Press, 1982.

[BLM91] C. Baral, J. Lobo, and J. Minker. WF^3: A Semantics for Negation in Normal Disjunctive Logic Programs with Equivalent Proof Methods. In *Proceedings of ISMIS 91*, pages 459–468, Dept of Computer Science, University of Maryland, October 1991. Springer-Verlag.

[BLM92] C. Baral, J. Lobo, and J. Minker. Generalized disjunctive well-founded semantics for logic programs. *Annals of Math and Artificial Intelligence*, 5:89–132, 1992.

[BM90] A. Bonner and L. McCarty. Adding negation as failure to intuitionistic logic programming. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proc. of the 1990 North American Conf.*, pages 681–703, 1990.

[BMPT92] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Meta for modularising logic programming. In *Proc. META-92*, pages 105–119, 1992.

[BNNS94] C. Bell, A. Nerode, R. Ng, and V.S. Subrahmanian. Computation and implementation of non-monotonic deductive databases. Technical Report CS-TR-2801, University of Maryland, 1991 (a revised version is to appear in JACM 94.

[BRSS92] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. The valid model semantics for logic programs. In *Proceedings of Principles of Database Systems*, pages 91–104, 1992.

[BS85] G. Bossu and P. Siegel. Saturation, nonmonotonic reasoning and the closed-world assumption. *Artificial Intelligence*, 25(1):13–63, 1985 January 1985.

[BS91] C. Baral and V. S. Subrahmanian. Duality between alternative semantics of logic programs and nonmonotonic formalisms. *In the International Workshop in logic programming and nonmonotonic reasoning - pages 69-86- and to appear in Journal of Automated Reasoning*, 1991.

[BS92] C. Baral and V. S. Subrahmanian. Stable and Extension Class Theory for Logic Programs and Default Logics. *Journal of Automated Reasoning*, 8:345–366, 1992.

[Cav89] L. Cavedon. Continuity, consistency, and completeness properties for logic programs. In Giorgio Levi and Maurizio Martelli, editors, *Logic Programming: Proc. of the Sixth Int'l Conf.*, pages 571–584, 1989.

[CDT91] L. Console, D. Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 2(5), 1991.

[CH85] A. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 2(1):1–5, 1985.

[Cha93] E. Chan. A Possible Worlds Semantics for Disjunctive Databases. *IEEE Transactions on Data and Knowledge Engineering*, 5(2):282–292, 1993.

[Che93] J. Chen. Minimal knowledge + negation as failure = only knowing (sometimes). In *Proceedings of the Second Int'l Workshop on Logic Programming and Nonmonotonic Reasoning, Lisbon*, pages 132–150, 1993.

[CKPR73] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un Systeme de Communication Homme-Machine en Francais. Technical report, Groupe de Intelligence Artificielle Universitae de Aix-Marseille II, Marseille, 1973.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[CL89] S. Costantini and G.A. Lanzarone. A metalogic programming language. In G. Levi and M. Martelli, editors, *Proc ICLP'89*, pages 218–233, 1989.

[CL90] M. Cadoli and M. Lenzerini. The Complexity of Closed World Reasoning and Circumscription. In *Proceedings AAAI-90*, pages 550–555, 1990. Full paper to appear in *Journal of Computer and System Sciences*.

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

[CP86] P. Cox and T. Pietrzykowski. Causes for events: their computation and applications. In J.H. Siekmann, editor, *Proc. of CADE-86*, pages 608–621, 1986.

[CS92] M. Cadoli and M. Schaerf. A survey on complexity results for nonmonotonic logics. Technical report, University di Roma "La Sapienza", Dipartiment o di Informatica e sistemistica, Roma, Italy, 1992.

[DDS92] M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for normal abductive logic programs. In *Proc. of JICSLP.*, pages 686–700, 1992.

[DDS93] M. Denecker and D. De Schreye. Representing incomplete knowledge in abductive logic programming, 1993. In *Proc. of the International Symposium on Logic Programming*, pages 147–163, 1993.

[DG84] W. Dowling and H. Gallier. Linear time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1:267–284, 1984.

[Dix91] J. Dix. Classifying semantics of logic programs. In *Proceedings of* International Workshop in logic programming and nonmonotonic reasoning*, Washington D.C.*, pages 166–180, 1991.

[Dix92] J. Dix. Classifying semantics of disjunctive logic programs. In *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 798–812, 1992.

[DMB92] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *Proc. of ECAI92*, 1992.

[Doy79] J. Doyle. A truth-maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Dun91a] P. Dung. Negation as hypotheses: an abductive foundation for logic programming. In *Proc. of ICLP-91*, pages 3–17, 1991.

[Dun91b] P. Dung. Well-founded reasoning with classical negation. In *Proc. of 1st international workshop on logic programming and non-monotonic reasoning*, 1991.

[Dun92] P. Dung. On the relations between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 105:7–25, 1992.

[Dun93a] P. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning and logic programming. In *Proc. of IJCAI 93*, pages 852–857, 1993.

[Dun93b] P. Dung. Representing actions in logic programming and its application in database updates. In *Proceedings of the International Conference in Logic Programming*, pages 222–238, 1993.

[EG92] T. Eiter and G. Gottlob. On the Complexity of Propositional Knowledge Base Revision, Updates, and Counterfactuals. *Artificial Intelligence*, 57(2–3):227–270, 1992.

[EG93a] T. Eiter and G. Gottlob. Complexity Aspects of Various Semantics for Disjunctive Databases. In *Proceedings of the Twelth ACM SIGACT SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-93)*, pages 158–167, June 1993.

[EG93b] T. Eiter and G. Gottlob. Complexity Results for Disjunctive Logic Programming and Application to Nonmonotonic Logics. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium (ILPS)*, Vancouver, 1993. MIT Press.

[EG93c] T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 1993. to appear.

[EG93d] T. Eiter and G. Gottlob. Propositional Circumscription and Extended Closed World Reasoning are $\mathbf{\Pi}_2^P$-complete. *Theoretical Computer Science*, 114(2):231–245, 1993. Addendum 118:315.

[EG93e] T. Eiter and G. Gottlob. The Complexity of Logic-Based Abduction. *Journal of the ACM*, 1993. To appear. Extended abstract in Proceedings STACS 1993, P. Enjalbert and A. Finkel and K. Wagner eds., Springer LNCS 665, pages 70–79, 1993.

[EGM93] T. Eiter, G. Gottlob, and H. Mannila. Expressive Power and Complexity of Disjunctive Datalog. In H. Blair, W. Marek, A. Nerode, and J. Remmel, editors, *Proceedings of the Second Workshop on Structural Complexity and Recursion-Theoretic Methods in Logic Programming*, Vancouver, October 29 1993. Cornell University, Mathematical Sciences Institute. Also available as CD-TR 93/51, Christian Doppler Lab for Expert Systems, TU Vienna.

[EK88] K. Eshghi and R. Kowalski. Abduction through deduction, 1988. unpublished paper.

[EK89] K. Eshghi and R. Kowalski. Abduction compared with negation as failure. In Giorgio Levi and Maurizio Martelli, editors, *Logic Programming: Proc. of the Sixth Int'l Conf.*, pages 234–255, 1989.

[EKP91] D. Etherington, S. Kraus, and D. Perlis. Nonmonotonicity and the scope of reasoning. *Artificial Intelligence*, 52(3):221–261, 1991.

[Elk90] C. Elkan. A rational reconstruction of non-monotonic truth maintenance systems. *Artificial Intelligence*, 43, 1990.

[Esg90] K. Esghi. Computing Stable Models by Using the ATMS. In *Proc. AAAI-90*, pages 272–277, 1990.

[Esh88] K. Eshghi. Abductive planning with event calculus. In *Proc. of the Fith ICLP.*, pages 562–579, 1988.

[Eva89] C. Evans. Negation-as-failure as an approach to the Hanks and McDermott problem. In *Proc. of the Second Int'l Symp. on Artificial Intelligence*, 1989.

[Fag90] F. Fages. Consistency of Clark's completion and existence of stable models. Technical Report 90-15, Ecole Normale Superieure, 1990.

[FBJ88] M. Fitting and M. Ben-Jacob. Stratified and Three-Valued Logic programming Semantics. In R.A. Kowalski and K.A. Bowen, editors, *Proc. 5^{th} International Conference and Symposium on Logic Programming*, pages 1054–1069, Seattle, Washington, August 15-19, 1988.

[FH89] Y. Fujiwara and S. Honiden. Relating the tms to autoepistemic logic. In *Proc. IJCAI-89*, pages 1199–1205, 1989.

[FH91] H. Fujita and R. Hasegawa. A model generation theorem prover in KL1 using a ramified stack algorithm. In *Eight International Conference in Logic Programming*, pages 535–548, 1991.

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

[Fit86] M. Fitting. Partial models and logic programming. *Theoretical Computer Science*, 48:229–255, 1986.

[Fit91] M. Fitting. Well-founded semantics, generalized. In *Proceedings of* International Symposium on Logic Programming*, San Diego*, pages 71–84, 1991.

[FL92] J. Fernandez and J. Lobo. A proof procedure for stable theories. Technical report, University of Maryland, CS-TR-3034, 1992.

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

[FM91] J. Fernandez and J. Minker. Bottom-up evaluation of hierarchical disjunctive deductive databases. In *Eight International Conference in Logic Programming*, pages 660–675, 1991.

[FM92] J. Fernández and J. Minker. Disjunctive deductive databases. In *International Conference on Logic Programming and Automated Reasoning: Lecture Notes in Artificial Intelligence 624*, pages 332–356, St. Petersburg, Russia, 1992. Springer-Verlag. Invited Paper.

[Gab85] D. Gabbay. Theoretical foundations for non-monotonic reasoning in expert systems. In Krzysztof R. Apt, editor, *Proc. of the NATO Advanced Study Institute on Logics and Models of Concurrent Systems*, pages 439–457, La Colle-sur-Loup, France, October 1985. Springer-Verlag.

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

[Gel89] M. Gelfond. Autoepistemic logic and formalization of commonsense reasoning. In M Reinfrank, Johan de Kleer, M Ginsberg, and Erik Sandewall, editors, *Non-Monotonic Reasoning: 2nd International Workshop (Lecture Notes in Artificial Intelligence 346)*, pages 176–186. Springer-Verlag, 1989.

[Gel91] M. Gelfond. Epistemic approach to formalization of commonsense reasoning. Technical Report TR-91-2, University of Texas at El Paso, 1991.

[Gel92a] A. Van Gelder. The well-founded semantics of aggregation. In *Proceedings of Principles of Database Systems*, pages 127–138, 1992.

[Gel92b] M. Gelfond. Logic programming and reasoning with incomplete information. *Annals of Mathematics and Artificial Intelligence*, 1992. To appear.

[Gin87] M. Ginsberg, editor. *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann, CA, USA, 1987.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D. Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 579–597, 1990.

[GL91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–387, 1991.

[GL92] M. Gelfond and V. Lifschitz. Representing actions in extended logic programs. In *Joint International Conference and Symposium on Logic Programming.*, pages 559–573, 1992.

[GLPT91] M. Gelfond, V Lifschitz, H. Przymusińska, and M. Truszczyński. Disjunctive defaults. In J. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Second Int'l Conf.*, pages 230–237, 1991.

[GM90] L. Giordano and A. Martelli. Generalized stable models, truth maintenance and conflict resolution. In *Proc. of the Seventh International*, pages 427–441. The MIT Press, 1990.

[GO92] L. Giordano and N. Olivetti. Negation as failure in intuitionistic logic programming. In *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 430–445. The MIT Press, 1992.

[Got92] G. Gottlob. Complexity Results for Nonmonotonic Logics. *Journal of Logic and Computation*, 2(3):397–425, June 1992.

[GP86] M. Gelfond and H. Przymusinska. Negation as failure: Careful closure procedure. *Artificial Intelligence*, 30(3):273–287, 1986.

[GP90] M. Gelfond and H. Przymusinska. Inheritance reasoning in autoepistemic logic. *Fundamenta Informaticae*, 13(4):403–445, 1990.

[GP91] M. Gelfond and H. Przymusińska. Definitions in epistemic specifications. In A. Nerode, W. Marek, and Subrahmanian V. S., editors, *Logic Programming and Non–monotonic Reasoning: Proc. of the First Int'l Workshop*, pages 245–259, 1991.

[GP93a] M. Gelfond and H. Przymusinska. Reasoning in open domains. In L. Pereira and A. Nerode, editors, *Proceedings of the Second International Workshop in Logic Programming and Nonmonotonic Reasoning*, pages 397–413, 1993.

[GP93b] M. Gelfond and H. Przymusinska. Stratification in extended logic programs, 1993. manuscript.

[GPP86] M. Gelfond, H. Przymusinska, and T. Przymusinski. The extended closed world assumption and its relation to parallel circumscription. In *Proc. of the fifth ACM Symposium on Principles of Database Systems*, pages 133–139, 1986.

[GPP89] M. Gelfond, H. Przymusinska, and T. Przymusinski. On the Relationship Between Circumscription and Negation as Failure. *Artificial Intelligence*, 38:75–94, 1989.

[GPP90] M. Gelfond, H. Przymusinska, and T. Przymusinski. On the relationship between CWA, Minimal Model, and Minimal Herbrand Model semantics. *International Journal of Intelligent Systems*, 5(5):549–565, 1990.

[Gre69] C. Green. Theorem Proving by Resolution as a Basis for Question - Answering Systems. In B. Meltzer D. Michie, editor, *Machine Intelligence 4*, pages 183–205. Edinburgh University Press, New York, 1969.

[GT93] M. Gelfond and B. Traylor. Representing null values in logic programs. In *Workshop on Logic Programming with incomplete information, Vancouver, BC.*, 1993.

[Hay73] P. Hayes. Computation and deduction. In *Proceedings of the Second Symposium on Mathematical Foundations of Computer Science*, pages 105–118. Czechoslovakia: Czechoslovakian Academy of Sciences, 1973.

[HL91] P. Hill and J. Lloyd. The godel report. Technical Report TR-91-02, University of Bristol, 1991.

[HM87] S. Hanks and D. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[Hor] J. Horty. Some direct theories of non-monotonic inheritance. In D. Gabbay and C. Hogger, editors, *Forthcoming in Handbook of Logic in AI and logic programming.*

[HP88] L. Henschen and H. Park. Compiling the GCWA in Indefinite Databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 395–438. Morgan Kaufmann Pub., Washington, D.C., 1988.

[IKH92] K. Inoue, M. Koshimura, and R. Hasegawa. Embedding negation as failure into a model generation theorem prover. In *The Eleventh International Conference on Automated Deduction*, 1992.

[Ino91] K. Inoue. Extended logic programs with default assumptions. In *Proc. of ICLP91*, 1991.

[Ino92a] K. Inoue. Studies on Abductive and Nonmonotonic Reasoning. PhD thesis, Kyoto University, 1992.

[IS91] N. Iwayama and K. Satoh. Computing abduction using the TMS. In *Proc. of ICLP 91*, pages 505–518, 1991.

[JK91] U. Junker and K. Konolige. Computing the extensions of autoepistemic and default logics with a truth maintenance systems. In *Proc. of AAAI 90*, pages 278–283, 1991.

[JLL83] J. Jaffar, J-L. Lassez, and J. Lloyd. Completeness of the negation as failure rule. In *Proc. of IJCAI-83*, volume 1, pages 500–506, 1983.

[KK91] R. Kowalski and J. Kim. A metalogic programming approach to multy-agent knowledge and belief. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press, 1991.

[KM90] A. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proc. of ECAI-90*, pages 385–391, 1990.

[KM91] A. Kakas and P. Mancarella. Stable theories for logic programs. In *Proc. of ISLP-91*, pages 88–100, 1991.

[kak92] A. Kakas and R. Kowalski and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–771, 1992.

[Kow74] R. Kowalski. Predicate logic as a programming language. *Information Processing 74*, pages 569–574, 1974.

[Kow79] R. Kowalski. *Logic for Problem Solving*. North-Holland, 1979.

[Kow90] R. Kowalski. Problems and promises of computational logic. In J. Lloyd, editor, *Computational Logic: Symposium Proceedings*, pages 80–95. Springer, 1990.

[KP87] P. Kolaitis and C. Papadimitriou. Why not negation by fixpoint? In *Proc. of PODS-87*, pages 231–239, 1987.

[KS86] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[KS90] R. Kowalski and F. Sadri. Logic programs with exceptions. In D. Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 598–613, 1990.

[KS91] D. Kemp and P. Stuckey. Semantics of logic programs with aggregates. In *Proceedings of International Symposium on Logic Programming*, pages 387–401, 1991.

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

[Kun89] K. Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7(3):231–245, 1989.

[Leh89] D. Lehmann. What does a conditional knowledge base entail? In *Proceedings of KR 89*, pages 212–221, 1989.

[Lif85a] V. Lifschitz. Closed-world data bases and circumscription. *Artificial Intelligence*, 27, 1985.

[Lif85b] V. Lifschitz. Computing circumscription. In *Proc. of IJCAI-85*, pages 121–127, 1985.

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

[Lif89] V. Lifschitz. Between circumscription and autoepistemic logic. In R. Brachman, H. Levesque, and R. Reiter, editors, *Proc. of the First Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 235–244, 1989.

[Lif90] V. Lifschitz. On open defaults. In J. Lloyd, editor, *Computational Logic: Symposium Proceedings*, pages 80–95. Springer, 1990.

[Lif91] V. Lifschitz. Nonmonotonic databases and epistemic queries: Preliminary report. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 381–386, Sydney, Australia, 1991.

[Lif93] V. Lifschitz. A language for representing actions. In *Working Papers of the Second Int'l Symp. on Logical Formalizations of Commonsense Knowledge*, 1993.

[Lin91] F. Lin. *A study of Nonmonotonic Reasoning*. PhD thesis, Stanford University, Department of Computer Science, 1991. Stanford TR – STAN-CS-91-1385.

[Llo87] J. Lloyd. *Foundations of Logic Programming*. Springer–Verlag, second edition, 1987.

[LM85] J. Lassez and M. Maher. Optimal fixedpoints of logic programs. *Theoretical Computer Science*, 39:15–25, 1985.

[LMR92] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of disjunctive logic programming*. The MIT Press, 1992.

[Lov87] D.W. Loveland. Near-Horn PROLOG. In J.-L. Lassez, editor, *Proc.* 4^{th} *International Conference on Logic Programming*, pages 456–459, 1987.

[LS89] F. Lin and Y. Shoham. Argument systems: a uniform basis for nonmonotonic reasoning. In R. Brachman, H. Levesque, and R. Reiter, editors, *Proc. of the First Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 245–255, 1989.

[LS90] F. Lin and Y. Shoham. Epistemic semantics for fixed-points nonmonotonic logics. In R. Parikh, editor, *Theoretical Aspects of Reasoning and Knowledge: Proc. of the Third Conf.*, pages 111–120, Stanford University, Stanford, CA, 1990.

[LS92] F. Lin and Y. Shoham. A logic of knowledge and justified assumptions. *Artificial Intelligence*, 57:271–290, 1992.

[LS93] V. Lifschitz and G. Schwarz. Extended logic programs as autoepistemic theories. In *Proceedings of the Second Int'l Workshop on Logic Programming and Non-monotonic Reasoning, Lisbon*, pages 101–114, 1993.

[LW92] V. Lifschitz and T. Woo. Answer sets in general nonmonotonic reasoning. In *Proc. of the Third Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 603–614, 1992.

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

[MB88] R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in PROLOG. In E.L. Lusk and R.A. Overbeek, editors, *Proc. of CADE 9*, pages 415–434, 1988.

[McC59] J. McCarthy. Programs with common sense. In *Proc. of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationery Office.

[McC80] J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1, 2):27–39,171–172, 1980.

[McC86] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986.

[McD82] D. McDermott. Nonmonotonic logic II: Nonmonotonic modal theories. *Journal of the ACM.*, 29(1):33–57, 1982.

[MD80] D. McDermott and J. Doyle. Nonmonotonic logic I. *Artificial Intelligence*, 13(1,2):41–72, 1980.

[MD93] D. De Schreye and M. Denecker. Representing incomplete knowledge in abductive logic programming, 1993. manuscript.

[MDS92a] B. Martens and D. De Schreye. A Perfect Herbrand Semantics for Untyped Vanilla Meta-Programming. In *Proc. of the Joint International Conference and Symposium on Logic Programming*. The MIT Press, 1992.

[MDS92b] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. Technical report, Department of Comp. Science, Katholieke Universiteit Leuven, Belgium, 1992.

[Met92] *Meta 92, Uppsala, Sweeden.* Springer Verlag, June 1992.

[MH69] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[Mil86] D. Miller. A theory of modules in logic programming. In *Proc. of IEEE Symposium on Logic Programming*, pages 106–114, 1986.

[Min82] J. Minker. On indefinite data bases and the closed world assumption. In *Proc. of CADE-82*, pages 292–308, 1982.

[Min88] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Pub., 1988.

[Min93] J. Minker. An overview of nonmonotonic reasoning and logic programming. *Journal of Logic Programming*, 17(2,3,4):95–126, 1993.

[MNR93] W. Marek, A. Nerode, and J. Remmel. A theory of nonmonotonic systems-II. *Annals of Mathematics and Artificial Intelligence*, 1993. To appear.

[Moo85] R. Moore. Semantical Considerations on Nonmonotonic Logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[Mor88] P. Morris. The anomalous extension problem in default reasoning. *Artificial Intelligence*, 35(3):383–399, 1988.

[MR90] J. Minker and A. Rajasekar. A Fixpoint Semantics for Disjunctive Logic Programs. *Journal of Logic Programming*, 9(1):45–74, July 1990.

[MR93] J. Minker and C. Ruiz. On extended disjunctive logic programs. In *ISMIS*, 1993. invited paper.

[MS89] W. Marek and V.S. Subrahmanian. The relationship between logic program semantics and non–monotonic reasoning. In G. Levi and M. Martelli, editors, *Proc. of the Sixth Int'l Conf. on Logic Programming*, pages 600–617, 1989.

[MT89] W. Marek and M. Truszczyński. Stable semantics for logic programs and default reasoning. In E. Lusk and R. Overbeek, editors, *Proc. of the North American Conf. on Logic Programming*, pages 243–257, 1989.

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

[MT93] W. Marek and M. Truszczyński. Reflexive autoepistemic logic and logic programming. In *Proceedings of the Second Int'l Workshop on Logic Programming and Non-monotonic Reasoning, Lisbon*, pages 115–131. MIT Press, 1993.

[Myc83] A. Mycroft. Logic programs and many valued logics. In *Proc. of the 1st STACS conference*, 1983.

[MZ82] J. Minker and G. Zanon. An Extension to Linear Resolution with Selection Function. *Information Processing Letters*, 14(3):191–194, June 1982.

[Nel49] D. Nelson. Constructible falsity. *Journal of Symbolic Logic*, 14:16–26, 1949.

[NMS91] A. Nerode, W. Marek, and V. S. Subrahamanian, editors. *Logic Programming and Non-monotonic Reasoning: Proceedings of the First International Workshop.* MIT Press, 1991.

[PA93] L. Pereira and J. Alferes. Optative reasoning with scenario semantics. In *Proceedings of ICLP 93, Hungary*, pages 601–619, 1993.

[PAA] L. Pereira, J. Alferes, and J. Aparicio. Nonmonotonic reasoning with logic programming. *Journal of Logic Programming*, 17(2,3,4):227–264,1993.

[PAA91a] L. Pereira, J. Aparicio, and J. Alferes. Contradiction removal within well-founded semantics. In Anil Nerode, Victor Marek, and Subramanian V. S., editors, *Logic Programming and Non–monotonic Reasoning: Proc. of the First Int'l Workshop*, pages 105–119. MIT Press, 1991.

[PAA91b] L. Pereira, J. Aparicio, and J. Alferes. Derivation procedures for extended stable models. In *Proceedings of IJCAI 91*, pages 863–870, 1991.

[PAA91c] L. Pereira, J. Aparicio, and J. Alferes. Non-monotonic reasoning with well-founded semantics. In *Proc. of the Eight International Logic Programming Conference*, pages 475–489, 1991.

[PAA92a] L. Pereira, J. Alferes, and J. Aparicio. Default theory for well founded semantics with explicit negation. In D. Pearce and G. Wagner, editors, *Logic in AI, Proc. of European Workshop JELLA'92 (Lecture Notes in AI, 633)*, pages 339–356, 1992.

[PAA92b] L. Pereira, J. Alferes, and J. Aparicio. Well founded semantics for logic programs with explicit negation. In *Proc. of European Conference on AI*, 1992.

[PC89] S. Pimental and J. Cuadrado. A truth maintenance system based on stable models. In *Proc. North American Conference on Logic Programming*, 1989.

[Pea87] J. Pearl. Embracing causality in formal reasoning. In *Proc. AAAI-87*, pages 360–373, 1987.

[PGA87] D. Poole, R. Goebel, and R. Aleliunas. Theorist: a logical reasoning system for defaults and diagnosis. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pages 331–352. Springer-Verlag, New York, 1987.

[PN93] L. Pereira and A. Nerode, editors. *Logic Programming and Non-monotonic Reasoning: Proceedings of the Second International Workshop.* MIT Press, 1993.

[Poo88] D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–48, 1988.

[Poo89] D. Poole. What the lottery paradox tells us about default reasoning. In R. Brachman, H. Levesque, and R. Reiter, editors, *Proc. of the First Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 333–340, 1989.

[PP88] H. Przymusinska and T. Przymusinski. Weakly Perfect Model Semantics for Logic Programs. In R.A. Kowalski and K.A. Bowen, editors, *Proc. 5^{th} International Conference and Symposium on Logic Programming*, pages 1106–1120, Seattle, Washington, August 15-19, 1988.

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

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

[PP92] H. Przymusinska and T. Przymusinski. Stationary default extensions. In *Proceedings of 4th International Workshop on Non-monotonic reasoning*, pages 179–193, 1992.

[PR93] J. Pinto and R. Reiter. Temporal reasoning in logic programming: A case for the situation calculus. In *Proceedings of 10th International Conference in Logic Programming, Hungary*, pages 203–221, 1993.

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

[Prz88b] T. Przymusinski. Perfect model semantics. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1081–1096, 1988.

[Prz89a] T. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proc. of Principles of Database Systems*, 1989.

[Prz89b] T. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[Prz89c] T. Przymusinski. Three-valued formalizations of non-monotonic reasoning and logic programming. In R. Brachman, H. Levesque, and R. Reiter, editors, *Proc. of the First Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 341–348, 1989.

[Prz89d] T. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 1989.

[Prz90a] T. Przymusinski. Extended stable semantics for normal and disjunctive programs. In D. Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 459–477, 1990.

[Prz90b] T. Przymusinski. Stationary semantics for disjunctive logic programs and deductive databases. In *Proceedings of North American Conference on Logic Programming*, pages 40–62, 1990.

[Prz91] T. Przymusinski. Stable semantics for disjunctive programs. *New generation computing*, 9(3,4):401–425, 1991.

[PW89] D. Pearce and G. Wagner. Reasoning with negative information 1 – strong negation in logic programming. Technical report, Gruppe fur Logic, Wissentheorie and Information, Freie Universitat Berlin, 1989.

[pei32] C. Pearce. Elements of Logic In: C. Hartshorne and P. Weiss, editors, Collected Papers of Charles Sanders Peirce, Volume II, Harvard University Press, Cambridge, Ma, 1932.

[Rei78] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 119–140. Plenum Press, New York, 1978.

[Rei80a] R. Reiter. Equality and domain closure in first-order databases. *JACM*, 27:235–249, 1980.

[Rei80b] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1,2):81–132, 1980.

[Rei82] R. Reiter. Circumscription implies predicate completion (sometimes). In *Proc. of IJCAI-82*, pages 418–420, 1982.

[Rei87a] R. Reiter. Nonmonotonic reasoning. *Annual Review of Computer Science*, 2:147–186, 1987.

[Rei87b] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[Ric74] B. Richards. A point of reference. *Synthese 28*, 28:431–445, 1974.

[RLM89] A. Rajasekar, J. Lobo, and J. Minker. Weak Generalized Closed World Assumption. *Journal of Automated Reasoning*, 5:293–307, 1989.

[RM89] R. Reiter and A. Mackworth. A logical framework for depiction and image interpretation. *Artificial Intelligence*, 41(2):125–156, 1989.

[RM90] A. Rajasekar and J. Minker. On stratified disjunctive programs. *Annals of Mathematics and Artificial Intelligence*, 1(1-4):339–357, 1990.

[Ros89] K. Ross. A procedural semantics for well founded negation in logic programming. In *Proc. of the eighth Symposium on Principles of Database Systems*, pages 22–34, 1989.

[RP91] W. Rodi and S. Pimentel. A nonmonotonic assumption-based tms using stable bases. In J. Allen, R. Fikes, and Erik Sandewall, editors, *Proc. of the Second Int'l Conf. on Principles of Knowledge Representation and Reasoning*, 1991.

[RT88] K. Ross and R. Topor. Inferring Negative Information from Disjunctive Databases. *Journal of Automated Reasoning*, 4(2):397–424, December, 1988.

[Sak89] C. Sakama. Possible model semantics for disjunctive databases. In *Proc. of the first international conference on deductive and object oriented databases*, pages 1055–1060, 1989.

[Sat87] T. Sato. On the consistency of first-order logic programs. Technical report, ETL, TR-87-12, 1987.

[Sch90] J. Schlipf. The expressive power of the logic programming semantics. In *Proc. of the Ninth Symp. on Principles of Database Systems*, pages 196–204, 1990.

[Sch91] G. Schwarz. Autoepistemic logic of knowledge. In Anil Nerode, Victor Marek, and Subramanian V. S., editors, *Logic Programming and Non–monotonic Reasoning: Proc. of the First Int'l Workshop*, pages 260–274, 1991.

[Sch92] J. Schlipf. Complexity and undecidability results in logic programming. In *Workshop on Structural Complexity and Recursion-theoretic methods in Logic Programming*, 1992.

[Sch93] J. Schlipf. Some remarks on computability and open domain semantics, 1993. manuscript.

[Scr59] M. Scriven. Truisms as the grounds for historical explanations. In P. Gardiner, editor, *Theories of History.* Free Press, New York, 1959.

[Scr63] M. Scriven. New issues in the logic of explanation. In S. Hook, editor, *Philosophy and History.* New York University Press, New York, 1963.

[Sha89] M. Shanahan. Prediction is deduction but explanation is abduction. In *Proc. of IJCAI-89*, pages 1055–1060, 1989.

[She88] J. Shepherdson. Negation in Logic Programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufman Pub., 1988.

[Sho67] J. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, Mass, 1967.

[SI92] K. Satoh and N. Iwayama. A correct goal-directed proof procedure for a general logic programs with integrity constraints. In *Proc. of the Third Int'l Workshop on Extensions of Logic Programming*, pages 19–34, 1992.

[SL88] B. Smith and D. Loveland. A Simple Near-Horn Prolog Interpreter. In R.A. Kowalski and K.A. Bowen, editors, *Proc.* 5^{th} *International Conference and Symposium on Logic Programming*, pages 794–809, Seattle, Washington, August 15-19, 1988.

[ST93] G. Schwarz and M. Truszczyński. Nonmonotonic Reasoning is Sometimes Simpler. In G. Gottlob, A. Leitsch, and D. Mundici, editors, *Proceedings of the Third Kurt Gödel Colloquium*, number 713 in LNCS, pages 313–325, Brno, Czech Republic, August 24–27 1993. Springer.

[SZ90] D. Sacca and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of PODS 1990*, pages 205–217, 1990.

[Tou86] D. Touretzky. *The mathematics of inheritance systems.* Morgan Kaufmann, Los Altos, Ca., 1986.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S. Tarnlund, editor, *Proc. 2nd international logic programming conference*, pages 127–138, Uppsala, Sweeden, 1984.

[TS86] H. Tanaki and T. Sato. OLD resolution with tabulation. In *Proc. of the Third International Conference on Logic Programming*, 1986.

[TS88] R. Topor and L. Sonenberg. On domain independent databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 217–240. Morgan Kaufmann, San Mateo, CA., 1988.

[Tur93] K. Turner. A monotonicity theorem for extended logic programs. In D. S. Warren, editor, *Proc. of 10th International Conference on Logic Programming*, pages 567–585, 1993.

[vEK76] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM.*, 23(4):733–742, 1976.

[VG88] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann, San Mateo, CA., 1988.

[VG89] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. of PODS-89*, pages 1–10, 1989.

[VGRS91] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[Wag90] K. Wagner. Bounded query classes. *SIAM J. Comp.*, 19(5):833–846, 1990.

[Wag91] G. Wagner. Logic programming with strong negation and inexact predicates. *Journal of Logic and Computation*, 1(6):835–861, 1991.

[Wag93] G. Wagner. Reasoning with inconsistency in extended deductive databases. In *Proc. of 2nd International workshop on Logic programming and non-monotonic reasoning*, 1993.

[War91] D. S. Warren. Computing the well-founded semantics of logic programs. Technical Report 91/12, CS dept. SUNY Stony Brook, 1991.

[WB93] C. Witteveen and G. Brewka. Skeptical reason maintenance and belief revision. *Artificial Intelligence*, 61:1–36, 1993.

[WC93] D. S. Warren and W. Chen. Query evaluation under well-founded semantics. In *Proc. of PODS 93*, 1993.

[Wit91] C. Witteveen. Skeptical reason maintenance system is tractable. In J. A, R. Fikes, and E. Sandewall, editors, *Proc. of KR'91*, pages 570–581, 1991.

[YH85] A. Yahya and L. Henschen. Deduction in non–horn databases. *Journal of Automated Reasoning*, 1(2):141–160, 1985.