

# Classical Negation in Logic Programs and Disjunctive Databases

**Michael Gelfond**

University of Texas at El Paso  
El Paso, Texas 79968

**Vladimir Lifschitz**

University of Texas at Austin  
Austin, Texas 78712

## Abstract

*An important limitation of traditional logic programming as a knowledge representation tool, in comparison with classical logic, is that logic programming does not allow us to deal directly with incomplete information. In order to overcome this limitation, we extend the class of general logic programs by including classical negation, in addition to negation-as-failure. The semantics of such extended programs is based on the method of stable models. The concept of a disjunctive database can be extended in a similar way. We show that some facts of commonsense knowledge can be represented by logic programs and disjunctive databases more easily when classical negation is available. Computationally, classical negation can be eliminated from extended programs by a simple preprocessor. Extended programs are identical to a special case of default theories in the sense of Reiter.*

## 1 Introduction

An important limitation of traditional logic programming as a knowledge representation tool, in comparison with classical logic, is that logic programming does not allow us to deal directly with incomplete information. A consistent classical theory partitions the set of sentences into three parts: A sentence is either provable, or refutable, or undecidable. A logic program partitions the set of ground queries into only two parts: A query is answered either *yes* or *no*. This happens because the traditional declarative semantics of logic programming automatically applies the *closed world assumption* to all predicates, and each ground atom that does not follow from the facts included in the program is assumed to be false. Procedurally, the query evaluation methods of logic programming give the answer *no* to every query that does not succeed; they provide no counterpart of undecidable sentences, which represent the incompleteness of information in classical axiomatic theories.

In order to overcome this limitation, we propose to consider “extended” logic programs, that contain *classical negation*  $\neg$  in addition to negation-as-failure *not*. General logic programs provide negative information implicitly, through closed-world reasoning; an extended 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*.<sup>1</sup>

A *general logic program* [Lloyd, 1984] can be defined as a set of rules of the form

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

where  $n \geq m \geq 0$ , and each  $A_i$  is an atom. The word “general” stresses the fact that such rules may contain negation, and, consequently, are more general than Horn clauses.

Recall that a *literal* is a formula of the form  $A$  or  $\neg A$ , where  $A$  is an atom. The negation sign in the negative literal  $\neg A$  represents classical negation, not negation-as-failure, so that expressions of the form *not*  $A$ , occurring in general logic programs, are not literals according to this definition.

An *extended logic program* is a set of rules of the form

$$L_0 \leftarrow L_1, \dots, L_m, \text{not } L_{m+1}, \dots, \text{not } L_n, \quad (1)$$

where  $n \geq m \geq 0$ , and each  $L_i$  is a literal.

The semantics of extended programs described below is an extension of the stable model semantics for general logic programs proposed in [Gelfond and Lifschitz, 1988]. The stable model semantics defines when a set  $S$  of ground atoms is a “stable model” of a given program. A “well-behaved” program has exactly one stable model, and the answer that such a program is supposed to return for a ground query  $A$  is *yes* or *no*, depending on whether  $A$  belongs to the stable model or not. (The existence of several stable models indicates that the program has several possible interpretations.) For an extended program, we will define when a set  $S$  of ground *literals* qualifies as its *answer set*. If the program doesn’t contain classical negation, then its answer sets are the same as its stable models. A “well-behaved” extended program has exactly one answer set, and this set is consistent. The answer that the program is supposed to return for a ground query  $A$  is *yes*, *no* or *unknown*, depending on whether the answer set contains  $A$ ,  $\neg A$ , or neither. The answer *no* corresponds to the presence of explicit negative information in the program.

Consider, for instance, the extended program  $\Pi_1$  consisting of just one rule:

$$\neg Q \leftarrow \text{not } P.$$

Intuitively, this rule means: “ $Q$  is false, if there is no evidence that  $P$  is true.” We will see that 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 don’t contain *not*:

$$\neg P \leftarrow, \quad P \leftarrow \neg Q$$

and

$$\neg P \leftarrow, \quad Q \leftarrow \neg P.$$

Let’s call them  $\Pi_2$  and  $\Pi_3$ . Each of the programs has a single answer set, but these sets are different. The answer set of  $\Pi_2$  is  $\{\neg P\}$ ; the answer set of  $\Pi_3$  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.) The language of extended programs includes classical negation, but not classical implication.

This approach has important computational advantages. We will see that, under rather general conditions, evaluating a query for an extended program can be reduced to evaluating two queries for a program that doesn't contain classical negation. Our extension of general logic programs hardly brings any new computational difficulties.<sup>2</sup>

The class of extended programs is of interest in connection with the problem of the relation between logic programming and nonmonotonic formalisms (see [Przymusinski, 1988] for an overview). As shown in [Bidoit and Froidevaux, 1987], general logic programs can be viewed as default theories in the sense of Reiter [1980]. A similar reduction is applicable to extended programs, and they turn out to be a notational variant of a natural, easily identifiable subset of default logic. We can say that the class of extended programs is the place where logic programming meets default logic halfway.

## 2 Answer Sets

The semantics of extended programs treats a rule with variables as shorthand for the set of its ground instances. It is sufficient then to define answer sets for extended programs without variables. This will be done in two steps.

First, consider extended programs without variables that, in addition, don't contain *not* ( $m = n$  in every rule (1) of the program). Extended programs without *not* correspond to the positive (Horn) case of traditional logic programming. Every such program will have exactly one answer set. This set will consist, informally speaking, of all ground literals that can be generated using

- (i) the rules of the program, and
- (ii) classical logic.

Obviously, there is only one case when a set of ground literals may logically entail a ground literal that doesn't belong to it: when it contains a pair of complementary literals  $A, \neg A$ . This observation suggests the following definition.

Let  $\Pi$  be an extended program without variables that doesn't contain *not*, and let  $Lit$  be the set of ground literals in the language of  $\Pi$ . The *answer set* of  $\Pi$  is the smallest subset  $S$  of  $Lit$  such that

- (i) for any rule  $L_0 \leftarrow L_1, \dots, L_m$  from  $\Pi$ , if  $L_1, \dots, L_m \in S$ , then  $L_0 \in S$ ;
- (ii) if  $S$  contains a pair of complementary literals, then  $S = Lit$ .

We will denote the answer set of a program  $\Pi$  that doesn't contain negation-as-failure by  $\alpha(\Pi)$ .

If  $\Pi$  is a *positive* program, i.e., a program containing neither *not* nor  $\neg$ , then condition (ii) is trivial, and  $\alpha(\Pi)$  is simply the minimal model of  $\Pi$ . It is also clear that the answer sets given

above for programs  $\Pi_2$  and  $\Pi_3$  are in agreement with this definition:

$$\alpha(\Pi_2) = \{\neg P\}, \quad \alpha(\Pi_3) = \{\neg P, Q\}.$$

Now let  $\Pi$  be any extended program without variables. By *Lit* we again denote the set of ground literals in the language of  $\Pi$ . For any set  $S \subset \text{Lit}$ , let  $\Pi^S$  be the extended program obtained from  $\Pi$  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,  $\Pi^S$  doesn't 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  $\Pi$ . In other words, the answer sets of  $\Pi$  are characterized by the equation

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

For instance, in order to check that  $\{\neg Q\}$  is an answer set of the program  $\Pi_1$  in the example above, we should construct the program  $\Pi_1^{\{\neg Q\}}$ . This program contains one rule,

$$\neg Q \leftarrow$$

(the result of deleting *not*  $P$  from the only rule of  $\Pi_1$ ). The answer set of this program is  $\{\neg Q\}$ , the set that we started with. Consequently, this is indeed an answer set of  $\Pi_1$ . It is easy to check that no other subset of literals has the same fixpoint property.

The answer sets of  $\Pi$  are, intuitively, possible sets of beliefs that a rational agent may hold on the basis of the information expressed by the rules of  $\Pi$ . If  $S$  is the set of ground literals that the agent believes to be true, then any rule that has a subgoal *not*  $L$  with  $L \in S$  will be of no use to him, and he will view any subgoal *not*  $L$  with  $L \notin S$  as trivial. Thus he will be able to replace the set of rules  $\Pi$  by the simplified set of rules  $\Pi^S$ . If the answer set of  $\Pi^S$  coincides with  $S$ , then the choice of  $S$  as the set of beliefs is “rational.”

We need to verify, of course, that the second, more general definition of answer sets, when applied to a program without *not*, is equivalent to the first definition. This is an immediate consequence of the fact that, for such  $\Pi$ ,  $\Pi^S = \Pi$ , so that the fixpoint condition (2) turns into  $S = \alpha(\Pi)$ .

On the other hand, if  $\Pi$  doesn't contain  $\neg$ , then  $\Pi^S$  is a positive program, and its answer set doesn't contain negative literals. Consequently, an answer set of a general logic program—a program without classical negation—is a set of *atoms*. The definition of an answer set coincides in this case with the definition of stable model given in [Gelfond and Lifschitz, 1988]. (Notice that the sign  $\neg$  stands there for negation-as-failure and thus corresponds to *not* in the notation of this paper.) We conclude that the answer sets of a *general* logic program are identical to its stable models. In this sense, the semantics of extended programs, applied to general programs, turns into the stable model semantics. But there is one essential difference: The absence of an atom  $A$  in a stable model of a general program represents the fact that  $A$  is false; the absence

of  $A$  and  $\neg A$  in an answer set of an extended program is taken to mean that nothing is known about  $A$ . We will return to this point in the next section, and then again in Section 6.

We think of answer sets as incomplete theories (rather than “three-valued models,” used, for instance, in [Fitting, 1985], [Przymusiński, 1989] and [Van Gelder *et al.*, 1990]). When a program has several answer sets, it is incomplete also in another sense—it has several different interpretations, and the answer to a query may depend on the interpretation.

An extended program is *contradictory* if it has an inconsistent answer set (that is, an answer set containing a pair of complementary literals). For instance, the program  $\Pi_4$ , consisting of the rules

$$P \leftarrow, \quad \neg P \leftarrow,$$

is contradictory. It is clear that a general logic program cannot be contradictory.

**Proposition 1.** *Every contradictory program has exactly one answer set—the set of all literals,  $Lit$ .*

This proposition shows that our approach to negation is different from the “paraconsistent” approach of [Blair and Subrahmanian, 1989].

**Proof.** It is clear from the definition of answer sets that any answer set containing a pair of complementary literals coincides with  $Lit$ . The fact that a contradictory program cannot have any other answer set is a consequence of the following lemma:

**Lemma 1.** *An extended logic program cannot have two answer sets of which one is a proper subset of the other.*

**Proof.** Let  $S, S'$  be answer sets of  $\Pi$ , and  $S \subset S'$ . Clearly,  $\Pi^{S'} \subset \Pi^S$ . It follows that  $\alpha(\Pi^{S'}) \subset \alpha(\Pi^S)$ , i.e.,  $S' \subset S$ . Consequently,  $S = S'$ .

Being noncontradictory doesn’t guarantee the existence of answer sets. This can be illustrated by any general logic program without stable models, such as  $P \leftarrow not\ P$ .

### 3 Representing Knowledge Using Classical Negation

Sometimes the use of negation-as-failure in logic programs leads to undesirable results that can be eliminated by substituting classical negation for it. We are indebted to John McCarthy for the following example. A school bus may cross railway tracks under the condition that there is no approaching train. This fact can be expressed by the rule

$$Cross \leftarrow not\ Train$$

if the absence of the atom *Train* in the database is interpreted as the absence of an approaching train. But this knowledge representation convention is unacceptable if information about the presence or absence of a train may be not available. If, for instance, *Train* is not included in the database because the driver’s vision is blocked, then we certainly don’t want the bus to cross tracks.

The situation will be different if classical negation is used:

$$Cross \leftarrow \neg Train.$$

Then *Cross* will not belong to the answer set unless the negative fact  $\neg Train$  is included.<sup>3</sup>

The difference between *not P* and  $\neg P$  in a logic program is essential whenever we cannot assume that the available positive information about *P* is complete, i.e., when the “closed world assumption” [Reiter, 1978] is not applicable to *P*. The closed world assumption for a predicate *P* can be expressed in the language of extended programs by the rule

$$\neg P(x) \leftarrow not\ P(x). \quad (3)$$

When this rule is included in the program, *not P* and  $\neg P$  can be used interchangeably in the bodies of other rules. Otherwise, we use *not P* to express that *P* is not known to be true, and  $\neg P$  to express that *P* is false.

For some predicates, the assumption opposite to (3) may be appropriate:

$$P(x) \leftarrow not\ \neg P(x). \quad (4)$$

For instance, the set of *terminal vertices* of a directed graph can be defined by the following program  $\Pi_5$ :

$$\begin{aligned} \neg Terminal(x) &\leftarrow Arc(x, y), \\ Terminal(x) &\leftarrow not\ \neg Terminal(x). \end{aligned}$$

For any predicate *P*, we are free to include either of the rules (3), (4) in the database, or to leave out both of them. Consider the following example. Jack is employed by Stanford University, and Jane is employed by SRI International:

$$\begin{aligned} Employed(Jack, Stanford) &\leftarrow, \\ Employed(Jane, SRI) &\leftarrow. \end{aligned}$$

Any employed individual has an adequate income:

$$Adequate-Income(x) \leftarrow Employed(x, y).$$

The answer set of the program with these 3 rules is

$$\{Employed(Jack, Stanford), Employed(Jane, SRI), Adequate-Income(Jack), Adequate-Income(Jane)\}.$$

This set contains no negative literals, and it doesn’t allow us to assert, for instance, that Jack is not employed by SRI. The claim that the employment information in the database is complete is expressed by the closed world assumption for *Employed*:

$$\neg Employed(x, y) \leftarrow not\ Employed(x, y).$$

Appending this rule to the program will add the literals

$$\neg \text{Employed}(\text{Jack}, \text{SRI}), \neg \text{Employed}(\text{Jane}, \text{Stanford})$$

to the answer set. If the available employment information is complete for Stanford, but not for SRI, a more restricted rule should be used instead:

$$\neg \text{Employed}(x, \text{Stanford}) \leftarrow \text{not}, \text{Employed}(x, \text{Stanford}).$$

The following example of the use of negation in the 1981 British Nationality Act is quoted in [Kowalski, 1989]: “After commencement no person shall have the status of a Commonwealth citizen or the status of a British subject otherwise than under this Act.” This statement, in essence, postulates the closed world assumption for some predicates. Kowalski remarks that there is no need to represent statements of this kind explicitly in a (general) logic program, because they are implicit in the semantics of the language. He seems to agree, however, that it may be desirable to permit predicates whose definitions are not assumed to be complete, and to require an “explicit declaration” whenever a completeness assumption is made. In the language of extended programs, such declarations are represented by rules of the form (3).

We do not commit ourselves to any particular use of the closed world assumption by deciding which of two opposite predicates will be represented by a predicate constant, and which one will be considered negative. In this sense, the language of extended programs is symmetric, like default logic [Reiter, 1980], autoepistemic logic [Moore, 1985] and formula circumscription [McCarthy, 1986]. On the contrary, the modification of our approach proposed in [Kowalski and Sadri, 1990] is not symmetric.

In Section 6 we say more on the use of the closed world assumption in extended programs.

Here is one more example in which both kinds of negation are used. College X uses the following rules for awarding scholarships to its students:

1. Every student with a GPA of at least 3.8 is eligible.
2. Every minority student with a GPA of at least 3.6 is eligible.
3. No student with a GPA under 3.6 is eligible.
4. The students whose eligibility is not determined by these rules are interviewed by the scholarship committee.

The rules are encoded in the following extended program:

$$\begin{aligned} \text{Eligible}(x) &\leftarrow \text{HighGPA}(x), \\ \text{Eligible}(x) &\leftarrow \text{Minority}(x), \text{FairGPA}(x), \\ \neg \text{Eligible}(x) &\leftarrow \neg \text{FairGPA}(x), \\ \text{Interview}(x) &\leftarrow \text{not } \text{Eligible}(x), \text{not } \neg \text{Eligible}(x). \end{aligned}$$

The last rule says:  $\text{Interview}(x)$ , if there is no evidence that  $\text{Eligible}(x)$  and there is no evidence that  $\neg \text{Eligible}(x)$ .

This program is to be used in conjunction with a database specifying the values of the extensional predicates *Minority*, *HighGPA* and *FairGPA*. Assume, for instance, that the following two facts are available about one of the students:

$$FairGPA(Ann) \leftarrow, \quad \neg HighGPA(Ann) \leftarrow .$$

The database contains no information about *Minority(Ann)*. (Ann is a minority student who, as a matter of principle, declined to state this fact on her application.)

The extended program  $\Pi_6$ , consisting of these 6 rules, has one answer set:

$$\{FairGPA(Ann), \neg HighGPA(Ann), Interview(Ann)\}. \quad (5)$$

## 4 Reduction to General Programs

Let  $\Pi$  be an extended program. For any predicate  $P$  occurring in  $\Pi$ , let  $P'$  be a new predicate of the same arity. The atom  $P'(\dots)$  will be called the *positive form* of the negative literal  $\neg P(\dots)$ . Every positive literal is, by definition, its own positive form. The positive form of a literal  $L$  will be denoted by  $L^+$ .  $\Pi^+$  stands for the general program obtained from  $\Pi$  by replacing each rule (1) by

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

For instance,  $\Pi_5^+$  is the program

$$\begin{aligned} Terminal'(x) &\leftarrow Arc(x, y), \\ Terminal(x) &\leftarrow \text{not } Terminal'(x). \end{aligned}$$

This is, of course, the usual definition of *Terminal* in the language of general logic programs.

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

**Proposition 2.** *A consistent set  $S \subset Lit$  is an answer set of  $\Pi$  if and only if  $S^+$  is an answer set of  $\Pi^+$ .*

In this sense, the mapping  $\Pi \mapsto \Pi^+$  reduces extended programs to general programs—even though  $\Pi^+$  gives no indication that  $P'$  represents, intuitively, the negation of  $P$ .<sup>4</sup>

The proof of Proposition 2 is based on two lemmas.

**Lemma 2.** *For any noncontradictory program  $\Pi$  that doesn't contain *not*,*

$$\alpha(\Pi^+) = \alpha(\Pi)^+.$$

**Proof.** Since  $\Pi$  is noncontradictory,  $\alpha(\Pi)$  is simply the set of literals that can be generated by applying (the ground instances of) the rules of  $\Pi$

$$L_0 \leftarrow L_1, \dots, L_m$$

as inference rules. Similarly,  $\alpha(\Pi^+)$  is the set of atoms that can be generated by applying the corresponding positive rules,

$$L_0^+ \leftarrow L_1^+, \dots, L_m^+.$$

It is clear that the atoms derivable using these positive rules are exactly the positive forms of the literals derivable using the rules of the original program.

**Lemma 3.** *For any contradictory program  $\Pi$  that doesn't contain not,  $\alpha(\Pi^+)$  contains a pair of atoms of the form  $A, (\neg A)^+$ .*

**Proof.** Consider the set  $S$  of ground literals that can be generated by applying the rules of  $\Pi$  as inference rules. Assume that  $S$  is consistent. Then  $S = \alpha(\Pi)$ , so that  $\alpha(\Pi)$  is consistent. But this is impossible, because  $\Pi$  is contradictory. Consequently,  $S$  is inconsistent. Let  $A, \neg A$  be two complementary literals that belong to  $S$ , i.e., can be derived using the rules of  $\Pi$  as inference rules. By applying the corresponding positive rules, we can derive  $A$  and  $(\neg A)^+$ . Consequently,  $A, (\neg A)^+ \in \alpha(\Pi^+)$ .

**Proof of Proposition 2.** Without loss of generality we can assume that  $\Pi$  doesn't contain variables. Let  $S$  be a consistent set of literals. By definition,  $S^+$  is an answer set of  $\Pi^+$  when

$$S^+ = \alpha((\Pi^+)^{S^+}),$$

which can be rewritten as

$$S^+ = \alpha((\Pi^S)^+).$$

Our goal is to prove the equivalence

$$S^+ = \alpha((\Pi^S)^+) \iff S = \alpha(\Pi^S). \quad (6)$$

Case 1:  $\Pi^S$  is noncontradictory. Then, by Lemma 2, the left-hand side of (6) is equivalent to  $S^+ = \alpha(\Pi^S)^+$ , which is clearly equivalent to the right-hand side of (6). Case 2:  $\Pi^S$  is contradictory. Since  $S$  is consistent, the right-hand side of (6) is false. By Lemma 3,  $\alpha((\Pi^S)^+)$  contains a pair  $A, (\neg A)^+$ . Since  $S$  is consistent,  $S^+$  cannot contain such a pair, so that the left-hand side of (6) is false as well.

**Corollary.** *If a set  $S \subset Lit$  is consistent, and  $S^+$  is the only answer set of  $\Pi^+$ , then  $S$  is the only answer set of  $\Pi$ .*

**Proof.** By Proposition 2,  $S$  is an answer set of  $\Pi$ , so we only need to show that  $\Pi$  has no other answer sets. Let  $S' \subset Lit$  be an answer set of  $\Pi$ . Since  $S$  is consistent,  $\Pi$  is noncontradictory (Proposition 1), so that  $S'$  is consistent, too. Then, by Proposition 2,  $(S')^+$  is an answer set of  $\Pi^+$ , so that  $(S')^+ = S^+$  and consequently  $S' = S$ .

The corollary shows how query evaluation procedures developed for general logic programs can be applied to extended programs. If  $\Pi^+$  is “well-behaved” (for instance, stratified), and its answer set doesn't contain a pair of atoms of the form  $A, (\neg A)^+$ , then  $\Pi$  is “well-behaved” also, and a literal  $L \in Lit$  belongs to the answer set of  $\Pi$  if and only if the ground atom  $L^+$  belongs to the answer set of  $\Pi^+$ . The condition  $L^+ \in \Pi^+$  can be verified, in principle, by a usual logic programming system. For instance, the fact that the literals included in (5) indeed belong to the answer set of  $\Pi_6$  can be confirmed by applying the Prolog query evaluation procedure to  $\Pi_6^+$  and to the queries

$$FairGPA(Ann), HighGPA'(Ann), Interview(Ann).$$

Queries with variables can be processed in a similar way.

**Remark.** The consistency assumption in the statements of Proposition 2 and the corollary is essential, even if  $\Pi$  is noncontradictory and  $\Pi^+$  is stratified. This is demonstrated by the following example  $\Pi_7$ :

$$\begin{aligned} P &\leftarrow \text{not } \neg P, \\ Q &\leftarrow P, \\ \neg Q &\leftarrow P. \end{aligned}$$

This program has no answer sets, although  $\Pi_7^+$  is a stratified program. The answer set of  $\Pi_7^+$  is the positive form of the inconsistent set  $\{P, Q, \neg Q\}$ .

## 5 Relation to Default Logic

The stable model semantics can be equivalently described in terms of reducing logic programs to a fixpoint nonmonotonic formalism—default logic, autoepistemic logic or introspective circumscription.<sup>5</sup> The extension discussed here can be reformulated as a reduction to any of these formalisms, too. In this section we show how the language of extended programs can be embedded into default logic.<sup>6</sup>

The review of default logic below is restricted to the case of quantifier-free defaults, sufficient for our purposes.<sup>7</sup>

A *default* is an expression of the form

$$F \leftarrow G : MH_1, \dots, MH_k, \quad (7)$$

where  $F, G, H_1, \dots, H_k$  ( $k \geq 0$ ) are quantifier-free formulas.  $F$  is the *consequent* of the default,  $G$  is its *prerequisite*, and  $H_1, \dots, H_k$  are its *justifications*. A *default theory* is a set of defaults.<sup>8</sup>

The operator  $\Gamma_D$  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 (7) of any default from  $D$ , if  $G \in \Gamma_D(E)$  and  $\neg H_1, \dots, \neg H_k \notin E$  then  $F \in \Gamma_D(E)$ ;
- (ii)  $\Gamma_D(E)$  is deductively closed.

$E$  is an *extension* for  $D$  if  $\Gamma_D(E) = E$ .

We will identify a rule

$$L_0 \leftarrow L_1, \dots, L_m, \text{not } L_{m+1}, \dots, \text{not } L_n$$

with the default

$$L_0 \leftarrow L_1 \wedge \dots \wedge L_m : \overline{ML_{m+1}}, \dots, \overline{ML_n}, \quad (8)$$

where  $\overline{L}$  stands for the literal complementary to  $L$ :  $\overline{\overline{A}} = \neg A$ ,  $\overline{\neg A} = A$ . 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 3.** *For any extended program  $\Pi$ ,*

- (i) *if  $S$  is an answer set of  $\Pi$ , then the deductive closure of  $S$  is an extension of  $\Pi$ ;*
- (ii) *every extension of  $\Pi$  is the deductive closure of exactly one answer set of  $\Pi$ .*

Thus the deductive closure operator establishes a 1–1 correspondence between the answer sets of a program and its extensions.

The proof of Proposition 3 is based on a few lemmas. We denote the deductive closure of a set of sentences  $E$  by  $Th(E)$ .

**Lemma 4.** *If a set of sentences  $E$  belongs to the range of  $\Gamma_\Pi$  for some extended program  $\Pi$ , then*

$$E = Th(E \cap Lit).$$

**Proof.** Let  $E = \Gamma_\Pi(E')$ . Then  $E$  is deductively closed, so that

$$Th(E \cap Lit) \subset Th(E) = E.$$

We'll prove the opposite inclusion by showing that  $Th(E \cap Lit)$  satisfies both closure conditions characterizing  $\Gamma_D(E')$ :

- (i) for any ground instance  $F \leftarrow G : MH_1, \dots, MH_k$  of a rule of  $\Pi$ , if  $G \in Th(E \cap Lit)$  and  $\neg H_1, \dots, \neg H_k \notin E'$  then  $F \in Th(E \cap Lit)$ ;
- (ii)  $Th(E \cap Lit)$  is deductively closed.

The second assertion is obvious. Assume that  $G \in Th(E \cap Lit)$  and  $\neg H_1, \dots, \neg H_k \notin E'$ . As we have seen,  $Th(E \cap Lit) \subset E$ , so that

$$G \in E = \Gamma_\Pi(E').$$

Consequently,

$$F \in \Gamma_\Pi(E') = E.$$

Since  $F$  is a literal, it follows that

$$F \in E \cap Lit \subset Th(E \cap Lit).$$

**Lemma 5.** *If  $S$  is an answer set of an extended program, then*

$$Th(S) \cap Lit = S.$$

**Proof.** Let  $S$  be an answer set of an extended program  $\Pi$ . If  $\Pi$  is contradictory, then, by Proposition 1,  $S = Lit$  and consequently  $Th(S) \cap Lit = Lit$ . If not, then  $S$  is a consistent set of ground literals, so that the ground literals that logically follow from  $S$  are precisely the elements of  $S$ .

**Lemma 6.** *For any extended program  $\Pi$  and any deductively closed set of sentences  $E$ ,*

$$\Gamma_{\Pi}(E) = Th(\alpha(\Pi^{E \cap Lit})).$$

**Proof.** Since every default in  $\Pi$  can be equivalently replaced by its ground instances, it can be assumed without loss of generality that  $\Pi$  doesn't contain variables. We will show first that

$$\Gamma_{\Pi}(E) \subset Th(\alpha(\Pi^{E \cap Lit})) \quad (9)$$

by proving that  $Th(\alpha(\Pi^{E \cap Lit}))$  satisfies both closure conditions characterizing  $\Gamma_{\Pi}(E)$ . Let (8) be the default corresponding to a rule from  $\Pi$ , such that

$$L_1 \wedge \dots \wedge L_m \in Th(\alpha(\Pi^{E \cap Lit})) \quad (10)$$

and

$$\neg \overline{L_{m+1}}, \dots, \neg \overline{L_n} \notin E. \quad (11)$$

From (10) and Lemma 5,

$$L_1, \dots, L_m \in Th(\alpha(\Pi^{E \cap Lit})) \cap Lit = \alpha(\Pi^{E \cap Lit}). \quad (12)$$

Since  $E$  is deductively closed, (11) implies

$$L_{m+1}, \dots, L_n \notin E.$$

Consequently, the rule  $L_0 \leftarrow L_1, \dots, L_m$  belongs to  $\Pi^{E \cap Lit}$ . In view of (12), it follows that

$$L_0 \in \alpha(\Pi^{E \cap Lit}) \subset Th(\alpha(\Pi^{E \cap Lit})).$$

Furthermore,  $Th(\alpha(\Pi^{E \cap Lit}))$  is deductively closed. The inclusion (9) is proved.

The opposite inclusion

$$Th(\alpha(\Pi^{E \cap Lit})) \subset \Gamma_{\Pi}(E)$$

will be proved if we show that

$$\alpha(\Pi^{E \cap Lit}) \subset \Gamma_{\Pi}(E),$$

because  $\Gamma_{\Pi}(E)$  is deductively closed. We will do that by proving that  $\Gamma_{\Pi}(E) \cap Lit$  satisfies both closure conditions for  $\alpha(\Pi^{E \cap Lit})$ . Let

$$L_1, \dots, L_m \in \Gamma_{\Pi}(E) \cap Lit \quad (13)$$

for some rule  $L_0 \leftarrow L_1, \dots, L_m$  of  $\Pi^{E \cap Lit}$ . According to the definition of  $\Pi^S$  (Section 2), this rule is obtained from a rule (7) of  $\Pi$  by deleting *not*  $L_{m+1}, \dots, \text{not } L_n$ , and, moreover,

$$L_{m+1}, \dots, L_n \notin E \cap Lit.$$

Since  $L_{m+1}, \dots, L_n$  are literals, it follows that

$$L_{m+1}, \dots, L_n \notin E.$$

By the definition of  $\Gamma_\Pi(E)$  applied to the default (8), (13) implies  $L_0 \in \Gamma_\Pi(E)$ , and consequently  $L_0 \in \Gamma_\Pi(E) \cap Lit$ . Furthermore, if  $\Gamma_\Pi(E) \cap Lit$  contains a pair of complementary literals, then it is inconsistent. Since  $\Gamma_\Pi(E)$  is deductively closed, it coincides with the set of all sentences, so that  $\Gamma_\Pi(E) \cap Lit = Lit$ .

**Proof of Proposition 3.** Let  $S$  be an answer set of  $\Pi$ . By Lemmas 6 and 5 and by the definition of answer set,

$$\Gamma_\Pi(Th(S)) = Th(\alpha(\Pi^{Th(S) \cap Lit})) = Th(\alpha(\Pi^S)) = Th(S).$$

To prove part (ii), consider an extension  $E$  of  $\Pi$ .  $E \cap Lit$  is an answer set of  $\Pi$ , because, by Lemmas 5 and 6,

$$\alpha(\Pi^{E \cap Lit}) = Th(\alpha(\Pi^{E \cap Lit})) \cap Lit = \Gamma_\Pi(E) \cap Lit = E \cap Lit.$$

On the other hand, by Lemma 4, the deductive closure of  $E \cap Lit$  is  $E$ . It remains to show that, for any answer set  $S$ ,  $Th(S) = E$  only if  $S = E \cap Lit$ . Assume  $Th(S) = E$ . Then, by Lemma 5,

$$E \cap Lit = Th(S) \cap Lit = S.$$

## 6 The Closed World Interpretation of General Logic Programs

Syntactically, general logic programs are a special case of extended programs. Moreover, the “canonical” model of a “well-behaved” general program (the unique stable model) is identical to its solution set as defined in Section 2. In spite of this, there is a semantic difference between a set of rules viewed as a *general* program, and the same set of rules viewed as an *extended* program. The absence of a ground atom  $A$  in the “canonical” model of a general program indicates that  $A$  is false in the model, so that the correct answer to the query  $A$  is *no*; the absence of  $A$  in the answer set of the same collection of rules treated as an extended program indicates that the answer to this query should be *unknown*.

For instance, the answer set of the program

$$\begin{aligned} Even(0) &\leftarrow, \\ Even(S(S(x))) &\leftarrow Even(x) \end{aligned}$$

is

$$\{Even(0), Even(S(S(0))), \dots\}.$$

Since this set contains neither  $Even(S(0))$  nor  $\neg Even(S(0))$ , the semantics of extended programs tells us that the answer to the query  $Even(S(0))$  is *unknown*—contrary to the intended meaning of this definition of *Even*.

This meaning can be formally expressed in the language of extended programs by adding the closed world assumption for *Even*:

$$\neg Even(x) \leftarrow not\ Even(x)$$

(Section 3). Then the solution set becomes

$$\{Even(0), \neg Even(S(0)), Even(S(S(0))), \neg Even(S(S(S(0)))), \dots\}.$$

This example suggests that an extended program “semantically equivalent” to a general program  $\Pi$  can be obtained from  $\Pi$  by adding the closed world assumption for each of its predicates. Define the *closed world interpretation*  $CW(\Pi)$  of a general program  $\Pi$  to be the extended program obtained from  $\Pi$  by adding the rules

$$\neg P(x_1, \dots, x_n) \leftarrow not\ P(x_1, \dots, x_n)$$

for all predicate constants  $P$  from the language of  $\Pi$ , where  $x_1, \dots, x_n$  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  $\Pi$  as we expect. Let  $Pos$  stand for the set of all positive ground literals in the language of  $\Pi$ .

**Proposition 4.** *If  $S$  is an answer set of a general logic program  $\Pi$ , then*

$$S \cup \{\neg A : A \in Pos \setminus S\} \tag{14}$$

*is an answer set of  $CW(\Pi)$ . Moreover, every answer set of  $CW(\Pi)$  can be represented in the form (14), where  $S$  is an answer set of  $\Pi$ .*

**Proof.** Without loss of generality, we can assume that  $\Pi$  doesn’t contain variables. Then the result  $\Pi'$  of replacing all rules in  $CW(\Pi)$  by their ground instances can be written as

$$\Pi \cup \{\neg A \leftarrow not\ A : A \in Pos\}.$$

Let  $S$  be an answer set of  $\Pi$ ; we need to show that (14) is then an answer set of  $\Pi'$ . Denote (14) by  $S'$ . By the definition of  $\Pi^S$  (Section 2),

$$(\Pi')^{S'} = \Pi^S \cup \{\neg A \leftarrow : A \in Pos \setminus S\}.$$

It follows that

$$\alpha((\Pi')^{S'}) = \alpha(\Pi^S) \cup \{\neg A : A \in Pos \setminus S\} = S \cup \{\neg A : A \in Pos \setminus S\} = S',$$

so that  $S'$  is a solution set of  $\Pi'$ .

To prove the second claim, take any answer set  $S'$  of  $\Pi'$ , and define  $S = S' \cap Pos$ . Then

$$\begin{aligned} S' &= \alpha((\Pi')^{S'}) = \alpha(\Pi^S) \cup \{\neg A : A \in Pos \setminus S'\} \\ &= \alpha(\Pi^S) \cup \{\neg A : A \in Pos \setminus S\}. \end{aligned}$$

It is clear that the first summand in this union is the positive part of  $S'$ , and the second summand is the negative part. We conclude, first, that

$$\alpha(\Pi^S) = S' \cap Pos = S,$$

i.e.,  $S'$  is an answer set of  $\Pi$ , and, second, that

$$\{\neg A : A \in Pos \setminus S\} = S' \setminus Pos,$$

and consequently

$$S \cup \{\neg A : A \in Pos \setminus S\} = S \cup (S' \setminus Pos) = (S' \cap Pos) \cup (S' \setminus Pos) = S'.$$

## 7 Classical Negation in Disjunctive Databases

The idea of rules with disjunctive heads has received much attention in recent years. Many attempts have been made to define a declarative semantics for “disjunctive logic programs,” or “disjunctive databases”; references can be found in [Przymusinski, 1990].

Consider a simple example. Jack is employed by Stanford University or by SRI International; any employed individual has an adequate income. It follows that Jack has an adequate income. It would be easy to formalize this instance of commonsense reasoning in classical logic, but it is not clear how to express the given facts by a logic program. The following “disjunctive database” can be used:

$$\begin{aligned} & \text{Employed}(\text{Jack}, \text{Stanford}) \mid \text{Employed}(\text{Jack}, \text{SRI}) \leftarrow, \\ & \text{Adequate-Income}(x) \leftarrow \text{Employed}(x, y). \end{aligned} \tag{15}$$

We use  $\mid$  rather than  $\vee$  in the head of the disjunctive rule, because there is a subtle difference, as we will see later, between the use of disjunction in the heads of rules and the use of disjunction in classical logic (similar to the difference between the noncontrapositive  $\leftarrow$  and the contrapositive classical implication, discussed in the introduction, or to the difference between *not* and  $\neg$ ).

An *extended disjunctive database* is a set of rules of the form

$$L_1 \mid \dots \mid L_k \leftarrow L_{k+1}, \dots, L_m, \text{not } L_{m+1}, \dots, \text{not } L_n,$$

where  $n \geq m \geq k \geq 0$ , and each  $L_i$  is a literal. The following definition of an answer set for extended disjunctive databases generalizes the definition given in Section 2.

First let  $\Pi$  be an extended disjunctive database without variables that doesn’t contain *not*. As in Section 2,  $Lit$  stands for the set of ground literals in the language of  $\Pi$ . An *answer set* of  $\Pi$  is any minimal subset  $S$  of  $Lit$  such that

- (i) for each rule  $L_1 \mid \dots \mid L_k \leftarrow L_{k+1}, \dots, L_m$  from  $\Pi$ , if  $L_{k+1}, \dots, L_m \in S$ , then, for some  $i = 1, \dots, k$ ,  $L_i \in S$ ;
- (ii) if  $S$  contains a pair of complementary literals, then  $S = Lit$ .

Consider, for instance, the database (15), with the second rule replaced by its ground instances. It has two answer sets:

$$\{\text{Employed}(\text{Jack}, \text{Stanford}), \text{Adequate-Income}(\text{Jack})\}$$

and

$$\{\text{Employed}(\text{Jack}, \text{SRI}), \text{Adequate-Income}(\text{Jack})\}.$$

The answer to a query may depend now on which answer set is selected. For instance, the answer to the query  $\text{Employed}(\text{Jack}, \text{Stanford})$  relative to the first answer set is *yes*, and the answer to the same query relative to the second set is *unknown*. The answer to  $\text{Adequate-Income}(\text{Jack})$  is unconditionally *yes*.<sup>9</sup>

Now let  $\Pi$  be any extended disjunctive database without variables. For any set  $S \subset Lit$ , let  $\Pi^S$  be the extended disjunctive database obtained from  $\Pi$  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,  $\Pi^S$  doesn't contain *not*, so that its answer sets are already defined. If  $S$  is one of them, then we say that  $S$  is an *answer set* of  $\Pi$ .<sup>10</sup>

In order to apply the definition of an answer set to an extended disjunctive database with variables, we first replace each rule by its ground instances.

As an example, let us add the closed world assumption for *Employed*

$$\neg \text{Employed}(x, y) \leftarrow \text{not } \text{Employed}(x, y)$$

to (15). The new database has two answer sets:

$$\{\text{Employed}(\text{Jack}, \text{Stanford}), \neg \text{Employed}(\text{Jack}, \text{SRI}), \text{Adequate-Income}(\text{Jack})\}$$

and

$$\{\text{Employed}(\text{Jack}, \text{SRI}), \neg \text{Employed}(\text{Jack}, \text{Stanford}), \text{Adequate-Income}(\text{Jack})\}.$$

It is interesting that the embedding of extended programs into default logic (Section 5) cannot be generalized to disjunctive databases in a straightforward way. Compare, for instance, the rule

$$P \mid Q \leftarrow \tag{16}$$

with the default

$$P \vee Q \leftarrow : . \tag{17}$$

The database (16) has two answer sets,  $\{P\}$  and  $\{Q\}$ ; the default theory (17) has one extension—the deductive closure of  $P \vee Q$ . This example shows that there is a difference between  $\mid$  as used in this paper and classical disjunction used in default logic.

As another example illustrating the difference between  $\mid$  and  $\vee$ , consider the database

$$\begin{aligned} Q &\leftarrow P, \\ P \mid \neg P &\leftarrow . \end{aligned} \tag{18}$$

Unlike the law of the excluded middle in classical logic, the second rule cannot be dropped without changing the meaning of the database. This rule expresses that  $P$  is either known to be true or known to be false. Any answer set of a database containing this rule includes either  $P$  or  $\neg P$ . The database consisting of just one rule  $Q \leftarrow P$  has one answer set, empty; (18) has two answer sets:  $\{P, Q\}$  and  $\{\neg P\}$ .

## 8 Conclusion

Extended logic programs and extended disjunctive databases use both classical negation  $\neg$  and negation-as-failure *not*. Their semantics is based on the method of stable models.

Some facts of commonsense knowledge can be represented by logic programs and disjunctive databases more easily when classical negation is available. In particular, rules of extended programs can be used for formalizing the closed world assumption for specific predicates.

Under rather general conditions, query evaluation for an extended program can be reduced to query evaluation for the general program obtained from it by replacing the classical negation of each predicate by a new predicate.

An extended program can be viewed as a default theory in which every justification and consequent is a literal, and every precondition is a conjunction of literals.

A semantic equivalent of a general logic program in the language of extended programs can be formed by adding the closed world assumption for all predicates.

## Acknowledgements

We are grateful to John McCarthy, Halina Przymusinska, Teodor Przymusinski and Rodney Topor for comments on earlier drafts. This research was supported in part by NSF grants IRI-8906516 and IRI-8904611 and by DARPA under Contract N00039-84-C-0211.

## Notes

1. The idea of providing for the incompleteness of information in logic-based query answering systems, and permitting answers other than simply *yes* or *no* for ground queries, is discussed in [Gelfond, 1989] and [Gelfond and Lifschitz, 1990]. Our use of two kinds of negation appears to be somewhat similar to the distinction between “strong” and “weak” negation in [Wagner, 1989].

2. Notice for comparison that when Poole and Goebel [1986] add classical negation to Prolog, they immediately get full first order logic and full resolution. This is because they interpret  $\leftarrow$  as classical implication.

3. According to [Kowalski, 1989] and [Kowalski and Sadri, 1990], many provisions in legislation have a negative form (“A declaration... shall not be registered unless...”). In a preliminary version of [Kowalski and Sadri, 1990], such statements were treated as integrity constraints with one of the conditions identified as “retractable.” The use of rules with negative heads seems more natural.

4. This mapping is related to the methods for translating inheritance hierarchies into logic programs proposed in [Gelfond and Lifschitz, 1989] and [Gelfond and Lifschitz, 1990]. These methods require that new predicates be introduced for the purpose of representing negative conditions. For instance, the assertion “penguins don’t fly” is written as

$$Flies'(x) \leftarrow Penguin(x),$$

where the positive literal  $Flies'(x)$  has, intuitively, the same meaning as the negative literal  $\neg Flies(x)$ . If we are willing to use the language of *extended* programs as the object language for this translation process, then the procedure can produce instead:

$$\neg Flies(x) \leftarrow Penguin(x).$$

The additional step of replacing  $\neg \textit{Flies}$  by  $\textit{Flies}'$  can be viewed as an instance of the general transformation  $\Pi \mapsto \Pi^+$ .

5. Historically, two of these reductions were proposed earlier than the definition of stable models. A reduction to autoepistemic logic is given in [Gelfond, 1987], and the equivalence of the stable model approach to this semantics is established in [Gelfond and Lifschitz, 1988] (Theorem 3). A reduction to default logic is described in [Bidoit and Froidevaux, 1987] (see also [Lin and Shoham, 1989]) and shown to be equivalent to stable models in [Bidoit and Froidevaux, 1988]. Introspective circumscription is defined in [Lifschitz, 1989], and its relation to stable models is discussed in Sections 5.1 and 5.2 of that paper.

6. The use of autoepistemic logic for this purpose leads to some complications. They are related to the fact that our semantics of extended programs is not contrapositive with respect to classical negation: as we have seen, the rule  $P \leftarrow \neg Q$  is generally not equivalent to  $Q \leftarrow \neg P$ . Consequently, even in programs without *not*, we can't interpret  $\leftarrow$  as implication. To fix this problem, we can try to modify the translation from [Gelfond, 1987] and insert the autoepistemic operator  $L$  in front of *each* literal in the body of a rule, even if it doesn't follow *not*. Then the rules  $P \leftarrow \neg Q$  and  $Q \leftarrow \neg P$  are translated by two different autoepistemic formulas,  $L\neg Q \supset P$  and  $L\neg P \supset Q$ . Unfortunately, this translation reduces some "well-behaved" programs to autoepistemic theories with several extensions. For instance, the autoepistemic theory  $LP \supset P$ , corresponding to the trivial program  $P \leftarrow P$ , has two extensions. The "unintended" extensions can be eliminated by using the ideas of [Marek and Truszczyński, 1989a] and [Marek and Truszczyński, 1989b].

7. This restriction allows us to disregard the process of Skolemization, involved in defining extensions in the general case ([Reiter, 1980], Section 7.1).

8. In the notation of [Reiter, 1980], (7) would be written as

$$\frac{G : MH_1, \dots, MH_k}{F},$$

or  $G : MH_1, \dots, MH_k / F$ . According to Reiter, a default theory may include, in addition to defaults, some formulas that play the role of axioms. However, this doesn't give any additional generality, because an axiom  $F$  can be identified with the default  $\textit{true} : / F$ .

9. For databases without classical negation, this definition is equivalent to the definition of a minimal model from [Minker, 1982].

10. This definition is roughly equivalent to the construction described in Section 6 of [Przymusiński, 1990], applied to the stable model semantics. (Contradictory programs are treated by Przymusiński somewhat differently.)

## References

- [Bidoit and Froidevaux, 1987] Nicole Bidoit and Christine Froidevaux. Minimalism subsumes default logic and circumscription. In *Proc. of LICS-87*, pages 89–97, 1987.
- [Bidoit and Froidevaux, 1988] Nicole Bidoit and Christine Froidevaux. Negation by default and nonstratifiable logic programs. Technical Report 437, Université Paris XI, 1988.

- [Blair and Subrahmanian, 1989] Howard Blair and V.S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.
- [Fitting, 1985] Melvin Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.
- [Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. of the Fifth Int’l Conf. and Symp.*, pages 1070–1080, 1988.
- [Gelfond and Lifschitz, 1989] Michael Gelfond and Vladimir Lifschitz. Compiling circumscriptive theories into logic programs. In Michael Reinfrank, Johan de Kleer, Matthew Ginsberg, and Erik Sandewall, editors, *Non-Monotonic Reasoning: 2nd International Workshop (Lecture Notes in Artificial Intelligence 346)*, pages 74–99. Springer-Verlag, 1989.
- [Gelfond and Lifschitz, 1990] Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int’l Conf.*, pages 579–597, 1990.
- [Gelfond, 1987] Michael Gelfond. On stratified autoepistemic theories. In *Proc. AAAI-87*, pages 207–211, 1987.
- [Gelfond, 1989] Michael Gelfond. Autoepistemic logic and formalization of commonsense reasoning. In Michael Reinfrank, Johan de Kleer, Matthew Ginsberg, and Erik Sandewall, editors, *Non-Monotonic Reasoning: 2nd International Workshop (Lecture Notes in Artificial Intelligence 346)*, pages 176–186. Springer-Verlag, 1989.
- [Kowalski and Sadri, 1990] Robert Kowalski and Fariba Sadri. Logic programs with exceptions. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int’l Conf.*, pages 598–613, 1990.
- [Kowalski, 1989] Robert Kowalski. The treatment of negation in logic programs for representing legislation. In *Proc. of the Second Int’l Conf. on Artificial Intelligence and Law*, pages 11–15, 1989.
- [Lifschitz, 1989] Vladimir Lifschitz. Between circumscription and autoepistemic logic. In Ronald Brachman, Hector Levesque, and Raymond Reiter, editors, *Proc. of the First Int’l Conf. on Principles of Knowledge Representation and Reasoning*, pages 235–244, 1989.
- [Lin and Shoham, 1989] Fangzhen Lin and Yoav Shoham. Argument systems: a uniform basis for nonmonotonic reasoning. In Ronald Brachman, Hector Levesque, and Raymond Reiter, editors, *Proc. of the First Int’l Conf. on Principles of Knowledge Representation and Reasoning*, pages 245–255, 1989.
- [Lloyd, 1984] John Lloyd. *Foundations of logic programming*. Springer-Verlag, 1984.

- [Marek and Truszczyński, 1989a] Wiktor Marek and Mirosław Truszczyński. Autoepistemic logic, defaults and truth maintenance. Manuscript, 1989.
- [Marek and Truszczyński, 1989b] Wiktor Marek and Mirosław Truszczyński. Relating autoepistemic and default logic. In Ronald Brachman, Hector Levesque, and Raymond Reiter, editors, *Proc. of the First Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 276–288, 1989.
- [McCarthy, 1986] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986. Reproduced in [McCarthy, 1990].
- [McCarthy, 1990] John McCarthy. *Formalizing common sense: papers by John McCarthy*. Ablex, Norwood, NJ, 1990.
- [Minker, 1982] Jack Minker. On indefinite data bases and the closed world assumption. In *Proc. of CADE-82*, pages 292–308, 1982.
- [Moore, 1985] Robert Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.
- [Poole and Goebel, 1986] David Poole and Randy Goebel. Gracefully adding negation and disjunction to Prolog. In Ehud Shapiro, editor, *Proc. of the Third Int'l Conf. on Logic Programming*, pages 635–641, 1986.
- [Przymusiński, 1988] Teodor Przymusiński. On the relationship between logic programming and non-monotonic reasoning. In *Proc. AAAI-88*, pages 444–448, 1988.
- [Przymusiński, 1989] Teodor Przymusiński. Three-valued formalizations of non-monotonic reasoning and logic programming. In Ronald Brachman, Hector Levesque, and Raymond Reiter, editors, *Proc. of the First Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 341–348, 1989.
- [Przymusiński, 1990] Teodor Przymusiński. Extended stable semantics for normal and disjunctive programs. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 459–477, 1990.
- [Reiter, 1978] Raymond Reiter. On closed world data bases. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 119–140. Plenum Press, New York, 1978.
- [Reiter, 1980] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.
- [Van Gelder *et al.*, 1990] Allen Van Gelder, Kenneth Ross, and John Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, pages 221–230, 1990.
- [Wagner, 1989] Gerd Wagner. The two sources of nonmonotonicity in vivid logic: weak falsity and inconsistency handling. In G. Brewka and H. Freitag, editors, *Proc. of the Workshop on Nonmonotonic Reasoning*, 1989.