Logic Programming and Knowledge Representation — the A-Prolog perspective.

Michael Gelfond [a] Nicola Leone [b]

[a] *Department of Computer Science,*
Texas Tech University
Lubbock, Texas, 79409-3104
mgelfond@cs.ttu.edu

[b] *Department of Mathematics,*
University of Calabria,
87030 Rende (CS), Italy,
leone@unical.it

Abstract

In this paper we give a short introduction to logic programming approach to knowledge representation and reasoning. The intention is to help the reader to develop a 'feel' for the field's history and some of its recent developments. The discussion is mainly limited to logic programs under the answer set semantics. For understanding of approaches to logic programming build on well-founded semantics, general theories of argumentation, abductive reasoning, etc., the reader is referred to other publications.

1 Introduction

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 update

⋆ The work of Michael Gelfond was partially supported by the NASA contract ncc9-143. The work of Nicola Leone was partially supported by the European Commission project ICONS, project no. IST-2001-32429.

both the knowledge base and the desired program behavior. A good knowledge representation language should allow construction of *elaboration tolerant* knowledge bases, i.e. bases in which small modifications of the informal body of knowledge correspond to small modifications of the formal base representing this knowledge. Around 1960, McCarthy [83], [84] proposed the use of *logical formulas* as a basis for a knowledge representation language of this type. It was soon suggested, however, that this tool is not always adequate [94]. This may be especially true in modeling commonsense behavior of agents when additions to the agent's knowledge are frequent and inferences are often based on the absence of knowledge. It seems that such reasoning can be better modeled by logical languages with nonmonotonic consequence relations which allow new knowledge to invalidate some of the previous conclusions. In precise terms a consequence relation $\models$ (over language $\mathcal{L}$) is called *nonmonotonic* if there are formulas A and B and a set of formulas T such that $T \models B$ and $T, A \not\models B$. Obviously the consequence relation of classical logic does not satisfy this property and is, therefore, monotonic.

The above observation has led to the development and investigation of new logical formalisms, *nonmonotonic logics*. The best known of them are circumscription [85,86], default logic [111], and nonmonotonic modal logics [89,88,95]. All of these logics are *super-classical*, i.e. can be viewed as an extension of classical predicate or propositional logic.

Another direction of research, started by Green [55], Hayes [56] and Kowalski [61], 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* [61,62,124] and the development of a logic programming language, Prolog [23].

Even though logic programming and nonmonotonic logic share many common goals and techniques, for some time there were no strong ties between the two research communities. Originally, 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 non-classical, 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*. The initial definition of this construct, incorporated in the original Prolog interpreter, 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 [30,110], was aimed at the development of a declarative semantics for logic programs with negation as failure. Further work in this direction proved to be fruitful for logic programming as well as for artificial intelligence and databases. The results uncovered deep similarities between various, seemingly different, approaches to formalization of nonmonotonic reasoning and shed a new light on the nature of rules and the negation as failure operator of Prolog. Among other things this lead to the development of the knowledge representation and reasoning language A-Prolog discussed in this paper and several other logic programming based languages with non-monotonic semantics [1,16,57,21,24,127].

Unlike Prolog these languages have well-defined declarative semantics independent of a particular inference mechanism. Unlike the 'original' nonmonotonic logics they are not super-classical. Instead they use a collection of new connectives which we believe are often more suitable for representing various forms of non-mathematical knowledge than their classical counterparts.

Papers published in this issue are selected from those presented at *LPNMR99* [1] – 5th International Conference on Logic Programming and Non-monotonic Reasoning, held in 1999 in El Paso, Texas. These papers are significant extensions of the respective versions presented at the conference. They deal with several different aspects of LPNMR: Knowledge Representation and Reasoning [72], Computational Complexity [54], Systems [118], Updates [2], Revision [76], and Applications [31]. The research work presented in [54] received the best paper award at LPNMR99.

This introductory paper is aimed at providing potential readers with some background information. This is not a survey of the field but rather a small collection of ideas and results put together to help the reader to develop a 'feel' for the field's history and some of its recent developments. There is by now a substantial number of books and surveys closely related to logic programming and nonmonotonic reasoning. Accurate mathematical exposition of the related formalisms can be found in [82], [17], [73], [74], [71]. For applications to various aspects of knowledge representation one can look at [51], [9], [116],[122], [1], [10], [32], [52],[28], [102], [39], [45], [100],[98],[115]. Issues related to reasoning methods of Prolog are discussed in [3], [101], [97]. Additional information and important historical perspective can be obtained from [91], [7], [92]. An in depth coverage of many aspects of knowledge representation and reasoning with A-Prolog can be found in the forthcoming book [8]; several logic-based works in artificial intelligence are collected in [93].

[1] The first LPNMR was organized in 1991 by Anil Nerode. Since that time the conference served as the main meeting place of people interested in both subjects.

The rest of the paper is organized as follows. Section 2 presents the syntax and semantics of A-Prolog and defines a couple of relevant syntactic fragments of it. Section 3 addresses computational aspects; it describes algorithms for reasoning associated with A-Prolog programs. Section 4 treats knowledge representation; it illustrates the basic methodology of representing knowledge in A-Prolog by examples. Section 5 highlights the relationship of A-Prolog to other nonmonotonic formalisms. Section 6 discusses the treatment of negation in logic programming. Section 7 analyzes the computational complexity of A-Prolog and its fragments, paying special attention to the impact of syntactic restrictions on negation and disjunction. Section 8 comments on some general properties of the entailment operators. Finally, Section 9 draws our conclusions.

2 The A-Prolog Language

We start with a description of syntax and semantics of A-Prolog (also called *Answer Set Programming* [81]) - a logic programming language based on answer sets/stable model semantics of [49,50].

2.1 Syntax

The syntax of A-Prolog is determined by a signature σ consisting of types, $types(\sigma) = \{\tau_0, \ldots, \tau_m\}$, object constants $obj(\tau, \sigma) = \{c_0, \ldots, c_m\}$ for each type τ, and typed function and predicate constants $func(\sigma) = \{f_0, \ldots, f_k\}$ and $pred(\sigma) = \{p_0, \ldots, p_n\}$. We will assume that the signature contains symbols for integers and for the standard functions and relations of arithmetic. Terms are built as in typed first-order languages; positive literals (or atoms) have the form $p(t_1, \ldots, t_n)$, where t's are terms of proper types and p is a predicate symbol of arity n; negative literals are of the form $\neg p(t_1, \ldots, t_n)$. The symbol $\neg$ is called *classical* or *strong* negation. [2] Literals of the form $p(t_1, \ldots, t_n)$ and $\neg p(t_1, \ldots, t_n)$ are called contrary. By $\bar{l}$ we denote a literal

[2] Logic programs with two negations appeared in [50] which was strongly influenced by the epistemic interpretation of logic programs given below. Under this view $\neg p$ can be interpreted as "believe that p is false" which explains the term "classical negation" used by the authors. A different view was advocated in [104,126] where the authors considered logic programs without negation as failure but with $\neg$. They demonstrated that in this context logic programs can be viewed as theories of a variant of intuitionistic logic with strong negation due to [96]. For more recent work on this subject see [106],[105]. We believe that both views proved to be fruitful and continue to play an important role in our understanding of A-Prolog. A somewhat different view on the semantics of programs with two negations can be found in [1].

contrary to l. Literals and terms not containing variables are called *ground.* The sets of all ground terms, atoms and literals over σ will be denoted by $terms(\sigma)$, $atoms(\sigma)$ and $lit(\sigma)$ respectively. For a set P of predicate symbols from σ, $atoms(P, \sigma)$ ($lit(P, \sigma)$) will denote the sets of ground atoms (literals) of σ formed with predicate symbols from P. Consistent sets of ground literals over signature σ, containing all arithmetic literals which are true under the standard interpretation of their symbols, are called *states* of σ and denoted by $states(\sigma)$.

A rule of A-Prolog is an expression of the form

$$l_0 \; or \; \ldots \; or \; l_k \leftarrow l_{k+1}, \ldots, l_m, not\; l_{m+1}, \ldots, not\; l_n \tag{1}$$

where l_i's are literals, *not* is a logical connective called *negation as failure* or *default negation*, and *or* is called *epistemic disjunction.* The following notation will be useful for further discussion. A set $\{not\; l_i, \ldots, not\; l_{i+k}\}$ will be denoted by $not\; \{l_i, \ldots, l_{i+k}\}$. If r is a rule of type (1) then $head(r) = \{l_0, \ldots, l_k\}, pos(r) = \{l_{k+1}, \ldots, l_m\}, neg(r) = \{l_{m+1}, \ldots, l_n\}$, and $body(r) = pos(r), not\; neg(r)$. A rule such that $head(r) = \emptyset$ is called a *constraint* and is written as

$$\leftarrow l_{k+1}, \ldots, l_m, not\; l_{m+1}, \ldots, not\; l_n \tag{2}$$

If $k = 0$ then we write

$$l_0 \leftarrow l_1, \ldots, l_m, not\; l_{m+1}, \ldots, not\; l_n \tag{3}$$

Default negation is interpreted as a new logical connective. Intuitively *not* l says that there is no reason to believe in l. Notice also the use of the symbol *or* instead of classical $\vee$. The meaning of *or* differs from that of $\vee$. A formula $A \vee B$ says that "A is *true* or B is *true*" while a rule, $A \; or \; B \leftarrow$, may be interpreted epistemically and means "A is believed to be *true* or B is believed to be *true*". (This approach can be viewed as a generalization of an early work by J. Minker [90].) A rule r such that $body(r) = \emptyset$ is called a *fact* and is often written as

$$l_0 \; or \; \ldots \; or \; l_k. \tag{4}$$

Definition 2.1 A *program* of A-Prolog is a pair $\{\sigma, \Pi\}$ where σ is a signature and Π is a collection of rules of the form (1). (In this paper we will often refer to programs of A-Prolog as *logic program* and denote them by their second element Π. The corresponding signature will be denoted by $\sigma(\Pi)$.)

2.2 Semantics

In our definition of semantics of A-Prolog we assume that the $l's$ in rule (1) are ground. Rules with variables (denoted by capital letters) will be used only as a shorthand for the sets of their ground instantiations. This approach is justified for the so called closed domains, i.e. domains satisfying the domain closure assumption [110] 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. Semantics of A-Prolog for open domains can be found in [9], [58].

The answer set semantics of a logic program Π assigns to Π a collection of *answer sets* – consistent sets of ground literals over signature $\sigma(\Pi)$ corresponding to beliefs which can be built by a rational reasoner on the basis of rules of Π. In the construction of these beliefs the reasoner is assumed to be guided by the following informal principles:

- He should satisfy the rules of Π, understood as constraints of the form: *If one believes in the body of a rule one must believe in at least one of the literals from the rule's head.*
- He should adhere to the *rationality principle* which says that *one shall not believe anything he is not forced to believe.*

The precise definition of answer sets will be first given for programs whose rules do not contain negation as failure. Let Π be such a program and let S be a state of $\sigma(\Pi)$.
We say that S is *closed* under Π if, for every rule

$$\{l_0, \dots, l_k\} \leftarrow l_{k+1}, \dots, l_m \tag{5}$$

of Π such that $\{l_{k+1}, \dots, l_m\} \subseteq S$, $\{l_0, \dots, l_k\} \cap S \neq \emptyset$. (Notice that for a constraint this condition means that the body is not contained in S.)

Definition 2.2 *(Answer set – part one)*
A state S of $\sigma(\Pi)$ is an *answer set* for Π if S is minimal (in the sense of set-theoretic inclusion) among the sets closed under Π. $\square$

It can be shown that a program without epistemic disjunction can have at most one answer set. To extend this definition to arbitrary programs, take any program Π, and let S be a state of $\sigma(\Pi)$. The *reduct*, Π^S, of Π relative to S is the set of rules

$$\{l_0, \dots, l_k\} \leftarrow l_{k+1}, \dots, l_m$$

for all rules (1) in Π such that $\{l_{m+1}, \dots, l_n\} \cap S = \emptyset$. Thus Π^S is a program without negation as failure.

Definition 2.3 *(Answer set – part two)*
A state S of $\sigma(\Pi)$ is an answer set for Π if S is an answer set for Π^S. $\Box$

(The above definition differs slightly from the original definition in [50], which allowed the inconsistent answer set, $lit(\sigma)$. Answer sets defined in this paper correspond to consistent answer sets of the original version.) Knowledge represented by programs of A-Prolog is frequently used for two different reasoning tasks, associated with two entailment relations defined below:

Definition 2.4 *(Entailment Relations)*

(1) A program Π *cautiously entails* a literal l ($\Pi \models l$) if l belongs to <u>all</u> answer sets of Π.
(2) A program Π *bravely entails* a literal l ($\Pi \models_b l$) if l belongs to <u>some</u> answer sets of Π.$\Box$

Obviously for programs having precisely one answer set, brave and cautious entailment coincide.

Some query answering systems for A-Prolog are based on the notion on cautious entailment; other use the brave one. Given a query l and program Π the cautious systems will first check if $\Pi \models l$. If this is the case the cautious answer to l will be *yes*; if $\Pi \models \bar{l}$ the cautious answer will be *no*; otherwise it will be *unknown*. In contrast, the brave systems attempt to find an answer set of Π containing l. If there is such an answer set, the brave answer to l will be *yes*; otherwise it will be *no*.

Example 2.1 Consider for instance a logic program

$$\Pi_0 \begin{cases} p(a) & \leftarrow not\ q(a). \\ p(b) & \leftarrow not\ q(b). \\ q(a). & \end{cases}$$

Using the definition of answer sets one can easily show that $S_0 = \{q(a), p(b)\}$ is an answer set of this program. In the next section we will introduce simple techniques which will allow us to show that it is the only answer set of Π_0. Thus $\Pi_0 \models q(a)$, $\Pi_0 \not\models q(b)$, $\Pi_0 \not\models \neg q(b)$ and Π_0's cautious answers to queries $q(a)$ and $q(b)$ will be *yes* and *unknown* respectively. The corresponding brave answers will be *yes* and *no*.

If we expand Π_0 by a rule

$$\neg q(X) \leftarrow not\ q(X) \tag{6}$$

the resulting program

$$\Pi_1 = \Pi_0 \cup (6)$$

would have the answer set $S = \{q(a), \neg q(b), p(b)\}$ and hence the cautious answer to query $q(b)$ will become *no*. The brave answer to $q(b)$ will not change. (Notice however that the brave answer to query $\neg q(b)$ will change from *no* to *yes*.)
Rule (6), read as "*if there is no reason to believe that X satisfies q then it does not*" is called the *closed world assumption* for relation q[110]. It guarantees that the reasoner's beliefs about q are complete, i.e. for any ground term t and every answer set S of the corresponding program, $q(t) \in S$ or $\neg q(t) \in S$.

It is worthwhile noting that the brave inference operator $\models_b$ may entail a literal l and its contrary $\bar{l}$.

Example 2.2 Consider the following program.

$$\begin{cases} p(a) \quad \leftarrow not\ \neg p(a). \\ \neg p(a) \leftarrow not\ p(a). \end{cases}$$

It is easy to see that the above program has two answer sets, namely, $\{p(a)\}$ and $\{\neg p(a)\}$. Thus, both $p(a)$ and $\neg p(a)$ are brave consequences of the program; while the program does not have any cautious consequence.

The cautious inference operator may entail, at the same time, a literal l and its contrary $\bar{l}$ <u>only</u> if the program does not have any answer set.

2.3 Program Properties

In this section we discuss several useful properties of logic programs. We hope that they help the readers to better understand the notion of answer set and to provide them with some insight into comparatively rich mathematical theory of A-Prolog.

2.3.1 The Basics

The following simple propositions ([77], [9], [66]) are frequently used to establish basic properties of logic programs.

Proposition 2.1 For any answer set S of a logic program Π:

(a) For any rule (1) from Π, if $\{l_{k+1}, \ldots, l_m\} \subseteq S$ and $\{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$ then there exists an i, $0 \leq i \leq k$ such that $l_i \in S$.

(b) If $l \in S$ then l is *supported* by Π; i.e., there exists a rule $r \in \Pi$ of the type (1) such that $\{l_{k+1}, \ldots, l_m\} \subseteq S$, $\{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$, and $\{l_0, \ldots, l_k\} \cap S = \{l\}$.

Proposition 2.2 For any program Π if S_0 and S_1 are answer sets of Π and $S_0 \subseteq S_1$ then $S_0 = S_1$.

Let us use these propositions to show that $S_0 = \{q(a), p(b)\}$ is the only answer set of program Π_0 from Example 2.1. Suppose S_1 is an answer set of Π_0. By Proposition 2.1 we have that $q(a) \in S_1$, $q(b) \notin S_1$, $p(b) \in S_1$, and hence, $S_0 \subseteq S_1$. By Proposition 2.2 we have that $S_0 = S_1$.

Programs of A-Prolog may have one, many, or zero answer sets. One can use the above propositions and the definition of answer sets to show that programs

$$\Pi_2 = \{p(a) \leftarrow not\ p(a).\}$$

and

$$\Pi_3 = \{p(a). \quad \neg p(a).\}$$

have no answer sets while program Π_4

$$\Pi_4 \begin{cases} e(0). \\ e(s(s(X))) \leftarrow not\ e(X). \\ p(s(X)) \quad\ \leftarrow e(X), not\ p(X). \\ p(X) \qquad\quad \leftarrow e(X), not\ p(s(X)). \end{cases}$$

has an infinite collection of them.

Finally, let us look at a few examples containing connectives *or* and $\neg$. It is easy to see that, due to the minimality condition in the definition of answer set, program

$$\Pi_5 = \{p(a)\ or\ p(b).\}$$

has two answer sets, $S_1 = p(a)$ and $S_2 = p(b)$. However, it will be wrong to view epistemic disjunction *or* as exclusive. We say that a conjunction $Q = l_1 \wedge \ldots \wedge l_n$ of literals is true in a set S if $l_1, \ldots, l_n \in S$; $\neg Q$ is true in S if for some i, $\overline{l_i} \in S$; otherwise Q is undefined in S. Obviously, neither $p(a) \wedge p(b)$

nor $\neg(p(a) \wedge p(b))$ holds in S_1, S_2 and therefore Π_5's answer to query Q will be *unknown*. It is instructive to contrast Π_5 with a program

$$\Pi_6 = \Pi_5 \cup \{\neg p(a) \ or \ \neg p(b)\}$$

which has answer sets $S_3 = \{p(a), \neg p(b)\}$ and $S_4 = \{p(b), \neg p(a)\}$ and clearly contains $\neg(p(a) \wedge p(b))$ among its consequences.

The notion of answer set is an extension of an earlier notion of stable model defined in [49] for *normal logic programs* (*nlp*). Syntactically, an *nlp* is simply a logic program consisting of rules of type (3) where l's are atoms. But, even though stable models of an *nlp* Π are identical to its answer sets, the meaning of Π under the stable model semantics is different from that under answer set semantics. The difference is caused by the closed world assumption 'hard-wired' in the definition of stable entailment $\models_s$: an *nlp* $\Pi \models_s \neg p(a)$ iff for every stable model S of Π, $p(a) \notin S$. In other words the absence of a reason for believing in $p(a)$ is sufficient to conclude its falsity. To match stable model semantics of Π in terms of answer sets, we need to expand Π by an explicit closed world assumption,

$$CWA(\Pi) = \Pi \cup \{\neg p(X) \leftarrow not \ p(X)\}$$

for every predicate symbol p of Π. Now it can be easily shown that for any ground literal l, $\Pi \models_s l$ iff $\Pi \models l$.

The next proposition (see [50], [9]) shows how programs of A-Prolog can be reduced to programs without $\neg$. 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 $p'(t_1, \ldots, t_n)$ will be called the *positive form* of the negative literal $\neg p(t_1, \ldots, t_n)$. Every positive literal is, by definition, its own positive form. The positive form of a literal l will be denoted by l^+. Π^+, called *positive form* of Π, stands for the normal logic program obtained from Π by replacing each rule (1) by

$$\{l_0^+, \ldots, l_k^+\} \leftarrow l_{k+1}^+, \ldots, l_m^+, not \ \ l_{m+1}^+, \ldots, not \ \ l_n^+$$

and adding the rules

$$\leftarrow p(t_1, \ldots, t_n), p'(t_1, \ldots, t_n)$$

for every atom $p(t_1, \ldots, t_n)$ of $\sigma(\Pi)$. For any set S of literals, S^+ stands for the set of the positive forms of the elements of S.

Proposition 2.3 A set $S \subset lit(\sigma(\Pi))$ is an answer set of Π if and only if S^+ is an answer set of Π^+. □

It is worthwhile noting that some answer set finders including **DLV** [27,42] and Smodels [118], [99], use the above rewriting technique to implement strong negation.

2.3.2 Some Syntactic Properties of Programs

In this section, we introduce two syntactically defined classes of logic programs with a number of useful and interesting properties. These and similar properties are often used for proving correctness of A-Prolog based knowledge and reasoning systems. First we need the following

Definition 2.5 Functions $|| \; ||$ from ground atoms of $\sigma(\Pi)$ to ordinals are called *level mappings* of Π.

Level mappings give us a useful technique for describing various classes of programs.

Definition 2.6 A logic program Π is called *(locally) stratified* [4], [107] if there is a level mapping $|| \; ||_s$ of Π such that for every rule r of Π

(1) For any $l \in pos(r)$, and for any $l' \in head(r)$, $||l||_s \leq ||l'||_s$;
(2) For any $l \in neg(r)$, and for any $l' \in head(r)$, $||l||_s < ||l'||_s$.

It is easy to see that program Π_0 from section 2 is stratified while programs Π_2 and Π_4 are not.

Theorem 2.1 A locally stratified normal program has exactly one answer set.

The theorem is an easy consequence of the results of [4], [107] which establish existence and uniqueness of the intended (perfect) model of locally stratified logic program and the results showing that perfect models of such programs coincide with their stable models. It is worthwhile noting that the above statement holds for *normal* logic programs; the presence of epistemic disjunction or the presence of strong negation invalidates the theorem. For instance, the locally stratified program $\{a \quad or \quad b\}$ has two answer sets (namely, $\{a\}$ and $\{b\}$); while the program $\{a, \neg a\}$, which also is locally stratified, does not have any answer sets at all. Theorem 2.1 is an example of a collection of results establishing existence and uniqueness of answer sets.

Another interesting class consists of head-cycle free programs.

Definition 2.7 A logic program Π is called *head-cycle free (hcf)* [11], if there is a level mapping $|| \; ||_h$ of Π such that for every rule r of Π

(1) For any $l \in pos(r)$, and for any $l' \in head(r)$, $||l||_h \leq ||l'||_h$;

(2) For any pair $l, l' \in head(r)$ $||l||_h \neq ||l'||_h$.

Example 2.3 Consider the following program Π_7.

$$\Pi_7 \begin{cases} a \ or \ b. \\ a \leftarrow b. \end{cases}$$

It is easy to see that Π_7 is head-cycle free. Consider now program

$$\Pi_8 = \Pi_7 \cup \{b \leftarrow a\}$$

Program Π_8 is not head-cycle free, since a and b should belong to the same level by Condition (1); while they cannot by Condition (2).

Among other things head-cycle free programs are interesting because epistemic disjunction can be safely eliminated from them by "shifting" some head atoms to the bodies of the rules. More precisely, by $sh(\Pi)$ we denote the disjunction-free program obtained from Π by substituting every rule of the form

$$a_1 \ or \ \ldots \ or \ a_k \leftarrow b_1, \ldots, b_m, not \ c_1, \ldots, not \ c_n$$

by the following k rules:

$$a_i \leftarrow b_1, \ldots, b_m, not \ c_1, \ldots, not \ c_n, not \ a_1, \ldots, not \ a_{i-1}, not \ a_{i+1}, \ldots, not \ a_k$$

where i ranges over interval $[1 \ldots k]$.

Theorem 2.2 *[11]* If Π is a head-cycle free program, then Π and $sh(\Pi)$ have exactly the same answer sets.

It is easy to see that the head-cycle free condition is essential: program Π_8 above has answer set $\{a, b\}$; while $sh(\Pi_8)$ has none. Later we will show that, in general, epistemic disjunction cannot be eliminated from A-Prolog without loss of the expressive power of the language.

3 Reasoning algorithms of A-Prolog

There are different systems which can be used for reasoning with programs of A-Prolog. The choice of the system normally depends on the form of the program and the type of queries one wants to be answered. Suppose for instance that our program Π has an infinite Herbrand universe and belongs to the class of so called *acyclic* programs [5]: A program is called acyclic if it has a level mapping $|| \ ||$ such that for any atom l occurring in the body of a rule with the head l_0, $||l_0|| > ||l||$. A normal acyclic logic program Π is stratified and therefore has unique answer set. It can be shown that various queries to Π can be answered by a variety of bottom-up evaluation algorithm (see for instance [5]). Moreover, acyclicity of R_0 together with some results from [6,120] guarantee that the SLDNF resolution based interpreter of Prolog will always terminate on atomic queries and produce the intended answers. Similar approximation of the A-Prolog entailment for a larger classes of programs with unique answer sets can be obtained by the system called *XSB* [21] implementing the well-founded semantics of [125]. Of course none of these traditional logic programming inference algorithms work for programs with multiple answer sets. Some algorithms addressing reasoning with such programs are based on the close relationship between answer sets and truth maintenance systems ([38], [37],[47]). In recent years however a number of substantially more efficient algorithms were developed to reason with programs with finite Herbrand universes, and a number of modern A-Prolog systems are now available. Two best known systems are among them **DLV** [27,42] and SMODELS [118]; but also other systems support A-Prolog to some extent, including CCALC [87], DCS [33], QUIP [34], and DeRes [22].

In this section, we briefly sketch one of such algorithms — the procecure underlying the computational engine of the **DLV** system [27,42]. Similar computational schemes are used by other answer set finding systems such as SMODELS [118].

The first subsection illustrates a procedure for the computation of an answer set of an A-Prolog program Π. The second subsection describes how such a procedure can be used for answering queries.

3.1 Computing an Answer Set

In this subsection, we describe a method for computing an answer set of a program Π which does not contain strong negation $\neg$. In the first step of the computation an A-Prolog system replaces a program Π, which normally contains variables, by its ground instantiation $ground(\Pi)$. It is worthwhile noting that $ground(\Pi)$ is not the full set of all syntactically constructible instances of the rules of Π; rather, it is an (often much smaller) subset of it having precisely the same answer sets as Π. The ability of the grounding procedure to construct small ground instantiation of the program may dramatically affect the performance of the entire system. As shown in [40], the adoption of database rewriting techniques has proved to be very useful for reducing the size of ground instantiation.

Once the variables have been eliminated from Π, the hard part of the computation is then performed on $\Pi_0 = ground(\Pi)$.

Function ModelGenerator(I: Interpretation): Boolean;
var inconsistency: Boolean;
begin
    I := DetCons(I);
    if I = lit(σ) **then return** *false*; (* inconsistency *)
    if no atom is undefined in I **then return** IsAnswerSet(I);
    else Select an undefined ground atom l according to a heuristic;
        if ModelGenerator($I \cup \{l\}$) **then return***true*;
        else return ModelGenerator($I \cup \{not\ l\}$);
end;

Fig. 1. Computation of Answer Sets

The heart of the computation is performed by the Model Generator, which is sketched in Figure 1. Roughly, the Model Generator produces some "candidate" answer sets of Π_0. The stability of each of them is subsequently checked by the function *IsAnswerSet(I)*, which verifies whether the given "candidate" I is a minimal model of the reduct Π_0^I of Π_0 relative to I. The function *IsAnswerSet(I)* returns *true* if the interpretation I at hand is an answer set and *false* otherwise (see [60] for details on this function).

The ModelGenerator is first called with parameter I set to the empty interpretation.[3] If the program Π has an answer set, then the function returns *true* setting I to the computed answer set; otherwise it returns *false*. The Model Generator is similar to the Davis-Putnam procedure employed by SAT solvers. It first calls a function DetCons(), which returns the extension of I with the literals that can be deterministically inferred from I (or the set of all literals lit(σ) upon inconsistency). This function (see [19,41] for details) is similar to a unit propagation procedure employed by SAT solvers, but exploits the peculiarities of A-Prolog for making further inferences (e.g., it exploits the knowledge that every answer set is a minimal model and must be supported). If DetCons does not detect any inconsistency, then an atom l is selected according to a heuristic criterion and ModelGenerator is (recursively) called on $I \cup \{l\}$ and on $I \cup \{not\ l\}$, to explore whether I can be extended to an answer set of Π. If one of such calls succeeds, then the function stops returning *true*, as an answer set of Π has been found. Upon failure of both such calls, the function returns *false*, as I cannot be extended to any answer set.

It is worthwhile remarking the importance of the criterion for choosing the

[3] An interpretation is a set of ground literals representing a 3-valued state of $\sigma(\Pi)$. An atom l can be *true* ($l \in I$), *false* ($not\ l \in I$) or *undefined* w.r.t. to an interpretation I. During the computation, undefinedness is eliminated to eventually converge to a 2-valued state.

atom l. The atom l plays the role of a branching variable of a SAT solver. And indeed, like for SAT solvers, the selection of a "good" atom l is crucial for the performance of an A-Prolog system. The adopted heuristic is one of the major differences between the existing A-Prolog systems, and often causes relevant performance gaps between them. An experimental analysis of a number of heuristics for A-Prolog systems can be found in [42].

3.2 Query Answering

The method for computing answer sets of A-Prolog programs without $\neg$, illustrated in the previous section, can be used to implement both brave and cautious reasoning with general A-Prolog programs.

Let L be a ground literal possibly preceded by the default negation *not*. By $\Pi(L)$ we denote the program Π with the addition of the following constraint: (i) $\leftarrow l$, if $L = not\ l$; (ii) $\leftarrow not\ l$, if $L = l$. The answer sets of $\Pi(L)$ are exactly the answer sets of Π where L happens to be true.

Let Π be an A-Prolog program (possibly containing $\neg$) and l be a ground literal. To answer the query l on Π, under brave and cautious entailments, one can proceed as follows.

Brave Reasoning Build the positive form $\Pi(l)^+$ of $\Pi(l)$.[4] Evaluate program $\Pi(l)^+$ as described in the previous subsection. If $\Pi(l)^+$ has an answer set, then l is a brave consequence of Π (i.e., $\Pi \models_b l$); otherwise, it is not.

Cautious Reasoning Build the positive form $\Pi(not\ l)^+$ and $\Pi(not\ \bar{l})^+$ of $\Pi(not\ l)$ and $\Pi(not\ \bar{l})$, respectively. Evaluate programs $\Pi(not\ l)^+$ and $\Pi(not\ \bar{l})^+$ as described in the previous subsection. If $\Pi(not\ l)^+$ does not have any answer set, then l is a cautious consequence of Π (i.e., $\Pi \models l$). If $\Pi(not\ \bar{l})^+$ does not have any answer set, then $\bar{l}$ is a cautious consequence of Π (i.e., $\Pi \models \bar{l}$).

4 A Simple Knowledge Base

To illustrate the basic methodology of representing knowledge in A-Prolog let us consider the following example:

[4] See Section 2.3 for the definition of positive form and for its properties.

Example 4.1 Let cs be a small computer science department located in the college of science, cos, of university, u. The department, described by the list of its members and the catalog of its courses, is in the last stages of creating its summer teaching schedule. In this example we outline a construction of a simple A-Prolog knowledge base $\mathcal{K}$ containing information about the department. For simplicity we assume an open-ended signature containing names, courses, departments, etc.

The list and the catalog naturally correspond to collections of atoms, say:

$$\begin{array}{llll} member(sam, cs). & member(bob, cs). & member(tom, cs). & \\ course(java, cs). & course(c, cs). & course(ai, cs). & course(logic, cs). \end{array} \tag{7}$$

together with the closed world assumptions expressed by the rules:

$$\begin{array}{ll} \neg member(P, cs) & \leftarrow not\ member(P, cs). \\ \neg course(C, cs) & \leftarrow not\ course(C, cs) \end{array} \tag{8}$$

The assumptions are justified by completeness of the corresponding information. The preliminary schedule can be described by the list, say:

$$teaches(sam, java). \quad teaches(bob, ai). \tag{9}$$

Since the schedule is incomplete the use of CWA for $teaches$ is not appropriate. The corresponding program correctly answers *no* to query '$member(mary, cs)$?' and *unknown* to query '$teaches(mary, c)$?'.

Let us now expand our knowledge base, $\mathcal{K}$, by the statement: 'Normally, computer science courses are taught only by computer science professors. The logic course is an exception to this rule. It may be taught by faculty from the math department.' This is a typical *default* with a *weak exception*[5] which can be represented in A-Prolog by the rules:

$$\begin{array}{ll} \neg teaches(P, C) & \leftarrow \neg member(P, cs), \\ & \quad course(C, cs), \\ & \quad not\ ab(d_1(P, C)), \\ & \quad not\ teaches(P, C). \\ ab(P, logic) & \leftarrow not\ \neg member(P, math). \end{array} \tag{10}$$

[5] An exception to a default is called *weak* if it stops application of the default without defeating its conclusion.

Here $d_1(P,C)$ is the name of the default rule and $ab(d_1(P,C))$ says that default $d_1(P,C)$ is not applicable to the pair $\langle P,C\rangle$. The second rule above stops the application of the default to any P who *may be* a math professor. Assuming that

$$member(mary, math). \tag{11}$$

is in $\mathcal{K}$ we have that $\mathcal{K}$'s answer to query '$teaches(mary,c)$?' will become *no* while the answer to query '$teaches(mary, logic)$?' will remain *unknown*. It may be worth noting that, since our information about persons membership in departments is complete, the second rule of 10 can be replaced by a simpler rule

$$ab(P, logic) \leftarrow member(P, math). \tag{12}$$

It is not difficult to show that the resulting programs have the same answer sets. To complete our definition of *teaches* let us expand $\mathcal{K}$ by the rule which says that 'Normally a class is taught by one person'. This can be easily done by the rule:

$$\begin{aligned} \neg teaches(P_1, C) \leftarrow\ & teaches(P_2, C), \\ & P_1 \neq P_2, \\ & not\ ab(d_2(C)), \\ & not\ teaches(P_1, C). \end{aligned} \tag{13}$$

Now if we learn that *logic* is taught by Bob we will be able to conclude that it is not taught by Mary.

The knowledge base $\mathcal{K}$ we constructed so far is elaboration tolerant with respect to simple updates. We can easily modify the departments membership lists and course catalogs. Our representation also allows strong exceptions to defaults, e.g. statements like

$$teaches(john, ai). \tag{14}$$

which defeats the corresponding conclusion of default (10). As expected, strong exceptions can be inserted in $\mathcal{K}$ without causing a contradiction.

Let us now switch our attention to defining the place of the department in the university. This can be done by expanding $\mathcal{K}$ by the rules

$$
\begin{aligned}
& part(cs, cos). \\
& part(cos, u). \\
& part(E1, E2) \quad \leftarrow part(E1, E), \\
& \qquad\qquad\qquad\quad\; part(E, E2). \\
& \neg part(E1, E2) \leftarrow not\; part(E1, E2).
\end{aligned}
\tag{15}
$$

$$
\begin{aligned}
member(P, E1) \leftarrow & \; part(E2, E1), \\
& \; member(P, E2).
\end{aligned}
\tag{16}
$$

The first two facts form a part of the hierarchy from the university organizational chart. The next rule expresses the transitivity of the *part* relation. The last rule of (15) is the closed world assumption for *part*; it is justified only if $\mathcal{K}$ contains a complete organizational chart of the university. If this is the case then the closed world assumption for *member* can be also expanded by, say, the rules:

$$\neg member(P, Y) \leftarrow not\; member(P, Y). \tag{17}$$

Let us now have a closer look at our program and see how theory of A-Prolog allows us to discover some of its interesting properties. First let us show that $\mathcal{K}$ has exactly one answer set, A_0.

Let $\mathcal{K}^+$ be the positive form of $\mathcal{K}$. It is easy to see that it is locally stratified and hence, by Theorem 2.1 has unique answer set, S^+. By Proposition 2.1 we conclude that there is no atom l such that l, $(\neg l)^+ \in S^+$. This implies that S is consistent and hence, by Proposition 2.3 is the only answer set of $\mathcal{K}$. This fact, together with Proposition 2.1 allows us to show that $\mathcal{K}$ will (correctly) entail that, say, *sam* is a member of the university, that u is not part of u, etc.

The answer set of $\mathcal{K}$ can be computed by the **DLV** system directly; some minor modifications are needed to run $\mathcal{K}$ on Smodels to enforce "domain restrictedness" (see [118]).

To check that *sam* is a member of the university we form a query

$$member(sam, u)? \tag{18}$$

Asking **DLV** to answer $member(sam, u)$? on program $\mathcal{K}$ under cautious entailment,[6] we get precisely the response to our query. **DLV** also provides simple means of displaying all the terms satisfying relations defined by a program and so we can use it to list, say, all members of the CS faculty, etc.

Readers with some knowledge of Prolog undoubtedly noticed that $\mathcal{K}$ is not suitable for the use with the Prolog interpreter. The program has a problem with left recursion in rule (15). In addition, Prolog interpreter will flounder[7] on a large number of queries to $\mathcal{K}$. Fortunately, floundering can be eliminated by the use of type predicates. A standard left recursion elimination applied to $\mathcal{K}$ will replace recursive rules of (15) by

$$\begin{aligned} part\_of(E1, E2) &\leftarrow part(E1, E2). \\ part\_of(E1, E2) &\leftarrow part(E1, E), \\ &\quad\; part\_of(E, E2). \\ \neg part\_of(E1, E2) &\leftarrow not\; part\_of(E1, E2). \end{aligned} \tag{19}$$

Using various termination and soundness and completeness results for Prolog type inference (see for instance [5], [6]) it is not difficult to show that if the transitive closure of our *part* relation is irreflexive then Prolog interpreter terminates and returns correct answer to queries formed by predicates $part\_of$ and *member*

Let us now expand $\mathcal{K}$ by a new relation, $offered(C, D)$, defined by the following, self-explanatory, rules:

$$\begin{aligned} offered(C, D) &\leftarrow course(C, D), \\ &\quad\; teaches(P, C). \\ \neg offered(C, D) &\leftarrow course(C, D), \\ &\quad\; not\; offered(C, D). \end{aligned} \tag{20}$$

Suppose also that either Tom or Bob are scheduled to teach the class in logic. A natural representation of this fact requires disjunction and can be expressed as

$$teaches(tom, logic)\; or\; teaches(bob, logic). \tag{21}$$

[6] In practice, this is done by adding $member(sam, u)$? to the file containing the program $\mathcal{K}$, and running it on **DLV** with option *-FC* to specify that cautious entailment is required.

[7] Prolog interpreter is said to flounder if during the execution it arrives at negative query containing variables. In Prolog floundering constitutes a serious programming error.

It is easy to see that the resulting program has two answer sets and that each answer set contains $offered(logic, cs)$. The corresponding reasoning can be done automatically by the **DLV** system. The example shows that A-Prolog with disjunction allows a natural form of reasoning by cases - a mode of reasoning not easily modeled by Reiter's default logic.

It is worth noting that this program is head-cycle free and therefore, by Theorem 2.2 the disjunctive rule (21) can be replaced by two non-disjunctive rules,

$$\begin{aligned} teaches(tom, logic) &\leftarrow not\ teaches(bob, logic). \\ teaches(bob, logic) &\leftarrow not\ teaches(tom, logic). \end{aligned} \tag{22}$$

and the resulting program will be equivalent to the original one. Now both, Smodels and **DLV** can be used to reason about the resulting knowledge base.

It is important to notice that development of an executable program by a series of transformation preserving the initial (possibly non-executable) specification is a standard programming methodology. The above example shows how declarativeness of A-Prolog and development of new reasoning algorithms allowed to shorten this transformation process and make programming easier.

Even though in the above example disjunction was eliminated by the simple transformation the complexity results [29] show that it is not always possible. Consider for instance the following example from [18].

Example 4.2 Suppose a holding owns some companies producing a set of products. Each product is produced by at most two companies. We will use a relation $produced\_by(P, C_1, C_2)$ which holds if a product P produced by companies C_1 and C_2. The holding below consists of four companies producing four products and can be represented as follows:

$$produced\_by(p1, b, s). \quad produced\_by(p2, f, b).$$
$$produced\_by(p3, b, b). \quad produced\_by(p4, s, p).$$

This slightly artificial representation, which requires a company producing a unique product to be repeated twice (as in the case of $p3$), is used to simplify the presentation.

Suppose also that we are given a relation $controlled\_by(C_1, C_2, C_3, C_4)$ which holds if companies C_2, C_3, C_4 control company C_1. In our holding, b and s control f, which is represented by $controlled\_by(f, b, s, s)$

Suppose now that the holding needs to sell some of the companies and that its policy in such situations is to maintain ownership of so called strategic companies, i.e. companies belonging to a minimal (with respect to the set theoretic inclusion) set S satisfying the following conditions:

(1) Companies from S produce all the products.
(2) S is closed under relation *controlled_by*, i.e. if companies C_2, C_3, C_4 belong to S then so is C_1.

It is easy to see that for the holding above the set $\{b, s\}$ is not strategic while the set $\{b, s, f\}$ is.

Suppose now that we would like to write a program which, given a holding of the above form, computes sets of its strategic companies. In A-Prolog this can be done as follows. Consider the rules

$$
\begin{array}{lll}
1.\ strat(C_1)\ or\ strat(C_2) & \leftarrow & produced\_by(P, C_1, C_2) \\
2.\ strat(C_1) & \leftarrow & controlled\_by(C_1, C_2, C_3, C_4), \\
 & & strat(C_2), \\
 & & strat(C_3), \\
 & & strat(C_4).
\end{array}
$$

defining the relation $strat(C)$ (C is strategic). Let Π be a program consisting of rules (1), (2) and an input database X of the type described above. The first rule guarantees that, for every answer set A of Π and every product p, there is a company c producing p such that an atom $strat(c) \in A$. The second rule ensures that for every answer set of Π the set of atoms of the form $strat(c)$ belonging to this set is closed under the relation *controlled_by*. Minimality of this set follows from the minimality condition in the definition of answer set. It is not difficult to check that answer sets of Π correspond one-to-one to strategic sets of the holding described by an input database. The **DLV** reasoning system can be asked to find an answer set of Π and display atoms of the form *strat* from it.

It is worthwhile noting that disjunction plays a crucial role in the above example; it is essential to encode that problem. The program is not head-cycle free, transforming disjunction to unstratified negation would alter the semantics of the program. Moreover, we cannot design at all another A-Prolog program encoding Strategic Companies without using disjunction. Indeed, since the Strategic Companies problem is Σ_2^P-hard [18], while normal logic programs can express "only" problems in NP (see Section 7), we can derive that Strategic Companies cannot be expressed by a fixed normal logic program uniformly on all collections of facts `produced_by`$(p, c1, c2)$ and `controlled_by`$(c, c1, c2, c3)$ (unless NP $= \Sigma_2^P$, an unlikely event) [18].

5 Logic programming and other nonmonotonic formalisms

Even though some affinity between logic programs and nonmonotonic logics was recognized rather early [112], [67], the intensive work investigating this phenomena started around 1987 after the discovery of model theoretic semantics for stratified logic programs [4]. Almost immediately after this notion was introduced, stratified logic programs were mapped into the three major nonmonotonic formalisms investigated at that time: circumscription [68],[107], autoepistemic logic [48] and default theories [12], [78]. Further work in this direction 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 and shed a new light on the nature of rules and negation as failure operator of Prolog. One of the results of this direction of research was the development of A-Prolog and several other knowledge representation languages with non-monotonic semantics. In this section, we will give some results establishing the relationship between A-Prolog and two other nonmonotonic logics.

5.1 A-Prolog and Autoepistemic Logic

We will start with an autoepistemic logic [95] whose formulas are built from propositional atoms using propositional connectives and the modal operator B.

Definition 5.1 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. $\square$

Intuitively, T is a set of axioms and E is a possible collection reasoner's beliefs determined by T. 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 [79]. We will denote this expansion by $Th(T)$.

Let us now consider a class G of programs of A-Prolog which consists of rules of the form:

$$\begin{array}{l} (i)\ \ p_0 \leftarrow p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n \\ (ii)\ \neg p \leftarrow \mathit{not}\ p \quad \text{(for every atom } p\text{).} \end{array} \tag{23}$$

where $0 < m \leq n$. Let α be a mapping which maps rules (i) and (ii) into autoepistemic formulas:

$$\begin{array}{l} (i) \;\; p_1 \wedge \ldots \wedge p_m \wedge \neg B\, p_{m+1} \wedge \ldots \wedge \neg B\, p_n \supset p_0 \\ (ii) \; \neg B\, p \supset \neg p \end{array} \tag{24}$$

and let

$$\alpha(\Pi) = \{\alpha(r) : r \in \Pi\}$$

Proposition 5.1 For any program $\Pi \in G$, 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 $\alpha(\Pi)$. Moreover, every stable expansion of $\alpha(\Pi)$ can be represented in the above form. □

Mapping α is a simple generalization of the mapping from [48] where 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 normal logic program Π and the stable expansions of $\alpha(\Pi)$. There are other interesting mappings of programs of A-Prolog into autoepistemic logic and its variants (see for instance [70], [80], [20], [117]) Even though these results substantially increase our understanding of the situation, none of the suggested mappings seem to provide a really good explanation of meaning of *or* and $\leftarrow$ connectives of A-Prolog in terms of autoepistemic logic.

5.2 A-Prolog and Reiter's Default Theories

A Reiter's *default* is an expression of the form

$$\frac{p : M\ j_1, \ldots, M\ j_n}{f} \tag{25}$$

where p, f and j's are quantifier-free first-order formulas;[8] f is called the *consequent* of the default, p is its *prerequisite*, and j's are its *justifications.* An expression $M\,j$ is interpreted as "it is consistent to believe j". A pair $\langle D, W\rangle$

[8] We limit ourselves to the quantifier-free case. For an interesting discussion on defaults with quantifiers see [69] and [59].

where D is a set of defaults and W is a set of first-order sentences is called Reiter's *default theory*.

Definition 5.2 Let $\langle D, W \rangle$ be a default theory and E be a set of first-order sentences. Consider $E_0 = W$ and, for $i \geq 0$, let D_i be the set of defaults of form (25) from D such that $p \in E_i$ and $\neg j_1 \notin E, \ldots, \neg j_n \notin E$. Finally, let $E_{i+1} = Th(E_i) \cup \{conseq(\delta) : \delta \in D_i\}$ where $Th(E_i)$ is the set of all classical consequences of E_i and $conseq(\delta)$ denotes the δ's consequent. The set E is called an *extension* for $\langle D, W \rangle$ if

$$E = \bigcup_{0}^{\infty} E_i$$

Extensions of a default theory D play a role similar to that of stable expansions of autoepistemic theories. The simple mapping α from programs of A-Prolog without disjunction to default theories identifies a rule, r

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n$$

with the default, $\alpha(r)$,

$$\frac{l_1 \wedge \ldots \wedge l_m : M\ \bar{l}_{m+1}, \ldots, M\ \bar{l}_n}{l_0} \tag{26}$$

(recall that $\bar{l}$ stands for the literal complementary to l).

Proposition 5.2 For any non-disjunctive program Π of A-Prolog

(i) if S is an answer set of Π, then $Th(S)$ is an extension of $\alpha(\Pi)$;
(ii) for every extension E of $\alpha(\Pi)$ there is exactly one answer set, S, of Π such that $E = Th(S)$

Thus, the class of non-disjunctive A-Prolog programs can be identified with the class of default theories with empty W and defaults of the form (26). This proposition from ([50]) is a simple extension of results from [12], and [78] where the authors considered this relationship for normal logic programs. Perhaps somewhat surprisingly, it is not easily generalized to program with disjunction. One of the problems in finding a natural translation from arbitrary A-Prolog programs to default theories is related to the inability to use defaults with empty justifications in reasoning by cases: The default theory with

$$D = \{ \frac{q :}{p}, \quad \frac{r :}{p} \}$$

and

$$W = \{q \vee r\}$$

does not have an extension containing p and therefore, does not entail p. The corresponding logic program

$p \leftarrow q$

$p \leftarrow r$

$q \ or \ r.$

has two answer sets, $\{p, q\}$ and $\{p, r\}$ and hence entails p.

6 A-Prolog and Negation in Logic Programs.

In this section we will briefly discuss the treatment of negation in logic programming. Let us start with definite programs, i.e. programs consisting of the rules

$$l_0 \leftarrow l_1, \ldots, l_m \tag{27}$$

where l's are atoms. Traditionally, such programs were viewed as (complete) definitions of objects and relations of the domain, and therefore, the lack of information about, say, truth of $p(a)$ was interpreter as evidence of its falsity. This is a familiar closed world assumption which, theoretically, can be formalized as an 'inference rule' of the form

$$\frac{\Pi \not\models l}{\neg l} \tag{28}$$

or equivalently

$$\frac{l \notin M_\Pi}{\neg l}$$

where M_Π is the minimal Herbrand model of Π. Of course the above statements do not really qualify as inference rules. First, their premises are not logical formulas, but the statements of the meta language. Second, since non-provability for definite programs is undecidable it is not always possible to determine if the rule is applicable or not. As a result a somewhat weaker version of CWA was implemented in Prolog: $\neg l$ is derivable from Π if the goal l has a *finitely failed SLD tree with respect to* Π. (For a definition of SLD trees and other related concepts see [3] or [101]).

$$\frac{l \text{ has a finitely failed SLD tree}}{\neg l} \tag{29}$$

To better understand the difference between the two let us consider a program $\Pi = \{p \leftarrow p\}$. It is easy to see that (28) entails $\neg p$ while (29) does not.

(The Prolog interpreter answering query p to Π will go into an infinite loop.) The difference can be used to divide the work on semantics of negation of Prolog into two parts. One approach attempts to formalize systems based on rule (28) and another is more interested in generalizations of rule (29). For simplicity we limit our discussion to semantics of normal logic programs, programs consisting of rules of the form

$$l_0 \leftarrow l_1, \ldots, l_m, \mathit{not}\ l_{m+1}, \ldots, \mathit{not}\ l_n \tag{30}$$

where l's are (not necessarily ground) atoms.

6.1 Clark's completion

The research on finding a declarative semantics for the *nlp* negation started with the pioneering work of Clark [30]. Given a *nlp* Π we can view the bodies of rules with a predicate p in their heads as "sufficiency" conditions for inferring p from the program. Clark suggested that the bodies of these rules 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, let us consider the following two step transformation of a *nlp* Π into a collection of first-order formula:

Step 1: Let $r \in \Pi$, $head(r) = p(t_1, \ldots, t_k)$, and $Y_1, \ldots, Y_s$ be the list of variable appearing in r. By $\alpha_1(r)$ we denote a formula:

$$\begin{aligned} &\exists\, Y_1 \ldots Y_s : X_1 = t_1 \wedge \ldots \wedge X_k = t_k \wedge \\ &\wedge l_1 \wedge \ldots \wedge l_m \wedge \neg l_{m+1} \wedge \ldots \wedge \neg l_n \ \supset\ p(X_1, \ldots, X_k) \end{aligned} \tag{31}$$

where, $X_1 \ldots X_k$ are variables not appearing in r.

$$\alpha_1(\Pi) = \{\alpha_1(r) : r \in \Pi\}$$

Step 2: For each predicate p if
$E_1 \supset p(X_1, \ldots, X_k)$
$\vdots$
$E_j \supset p(X_1, \ldots, X_k)$
are all the implications in $\alpha_1(\Pi)$ with p in their conclusions then replace these formulas by

$$\forall\, X_1 \ldots X_k : p(X_1, \ldots, X_k) \equiv E_1 \vee \ldots \vee E_j$$

if $j \geq 1$ and by

$$\forall\, X_1 \ldots X_k : \neg q(X_1, \ldots, X_k)$$

if $j = 0$.

Definition 6.1 The resulting first-order theory combined with *free equality axioms* from [30] is called *Clark's completion* of Π and is denoted by $Comp(\Pi)$. A literal l is *entailed* by Π if $l \in Th(Comp(\Pi))$.

The following theorem [4] establishes the relationship between models of Clark's completion of Π and the notion of *supported model.* (A set S of atoms is supported by Π if, for every $l \in S$, there is a rule (30) such that $l_1, \ldots, l_m \in S$ and $l_{m+1}, \ldots, l_n \notin S$.)

Theorem 6.1 *A set S of atoms is a model of Clark's completion of Π iff S is supported and closed under the rules of Π.*

Models of Clark's completion may obviously differ from answer sets of Π. Program $p \leftarrow p$ has two Clark's models, $\{\ \}$ and $\{p\ \}$, but only one answer set $\{\ \}$. This shall not be surprising — the completion semantics intends to capture the notion of finite failure of a particular inference mechanism, SLDNF resolution, while answer sets semantics formalizes more general notion of default negation. It is also important to note that the above theorem immediately implies that every literal entailed by Π with respect to the Clark's semantics is also entailed by Π with respect to the answer set semantics.

The existence of Clark's declarative semantics facilitated the development of the theory of logic programs. It made possible first proofs of correctness of inference mechanism based on SLDNF resolution, and of certain transformations of logic programs such as fold/unfold [121], proofs of equivalence and other properties of programs. It is still widely and successfully used for logic programming applications. Unfortunately in many situations the Clark's semantics appears too weak. Consider for instance the following example:

Example 6.1 Suppose that we are given a graph, say,

$$edge(a,b). \quad edge(c,d). \quad edge(d,c).$$

and want to describe vertices of the graph reachable from a given vertex a. The natural solution seems to be to introduce the rules:

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

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

$$reachable(X) \equiv (X = a \lor \exists\, Y : reachable(Y) \land edge(Y,X))$$

from which such a conclusion cannot be derived.

The difficulty was recognized as serious and prompted the development of other logic programming semantics, including that of the A-Prolog. Even though now there are comparatively few knowledge representation languages which use Clark's completion as the basis for their semantics the notion didn't loose its importance. As an illustration let us consider its use for computing answer sets of logic programs. We will need the following terminology.

Definition 6.2 A *nlp* Π is called *tight* if there is a level mapping $|| \; ||$ of Π such that for every rule (30) of Π

$$||l_0|| > ||l_1||, \ldots, ||l_m|| \tag{32}$$

Theorem 6.2 *If* Π *is tight then* S *is a model of* $Comp(\Pi)$ *iff* S *is an answer set of* Π.

The above theorem is due to F. Fages [43]. There are some recent results extending the notions of Clark's completion and of tightness, and discovering more general conditions for equivalence of the two semantics. Note that whenever the two semantics of Π are equivalent, Π's answer sets can be computed by a satisfiability solver which, in some cases, can be more efficient than the direct use of Smodels or **DLV**. More on this work can be found in [14].

6.2 Three-valued approaches

There were several important modifications of the Clark's semantics which are based on the use of three-valued logic. The first such modification [44,63,64], aimed at capturing finite failure with respect to SLDNF resolution, uses three-valued completion of a program. The following example illustrates the difference between two-valued and three valued completions:

Example 6.2 Consider program Π_9:

$$p \leftarrow not\ p.$$

$$q. \quad .$$

It is easy to see that $COMP(\Pi_9)$ is inconsistent while three valued completion is consistent and has a unique model in which p is *undefined* and q is *true*. This corresponds to the behavior of the SLDNF resolution which answers *yes* to q and goes into the loop on query p.

SLDNF resolution is sound with respect to the three valued completion. Unfortunately, it can be incomplete, but as shown in [26] the only sources of incompleteness are floundering and unfair selection of literals in the SLDNF derivation. Since there are multiple sufficient conditions for avoiding floundering three-valued completion and SLDNF resolution seem to be a good match.

The well-founded semantics of [125] formalizes negation viewed as (a not necessarily finite) failure. To give a precise definition we need the following terminology.

For any *nlp* Π, the function from sets of literals to sets of literals is defined by the equation

$$\gamma_\Pi(X) = \Pi^X \tag{33}$$

where Π^X is the reduct from definition 2.3 of answer set. It is clear that the answer sets of Π can be characterized as the fixpoints of γ_Π. It is not difficult to show that, if $X \subset Y$ then $\gamma_\Pi(Y) \subset \gamma_\Pi(X)$. This implies that the function γ_Π^2 is monotone and hence has the least fixpoint. Atoms belonging to this fixpoint are called *well-founded* relative to Π. Atoms belonging to the complement of the greatest fixpoint of γ_Π^2 are called *unfounded* relative to Π.

Definition 6.3 A three-valued interpretation which assigns 1 (*true*) to atoms well-founded relative to Π, 0 (*false*) to atoms unfounded relative to Π, and 1/2 (*undefined*) to all the remaining atoms is called the well-founded model of Π. A literal l is a well-founded consequence of Π if it is true in Π's well-founded model.

From the above definition one can easily see that every *nlp* has the well-founded model and that every well-founded consequence of Π is also Π's consequence with respect to the stable model semantics. To better understand the difference between the semantics let us look at several examples.

Example 6.3 Consider the program Π_9 from Example 6.2. It has no stable model (and hence Π_9's set of stable consequences consists of $\{p, \neg p, q, \neg q\}$). In contrast, the only well-founded consequence of Π_9 is q. The set of unfounded atoms is empty and the only undefined atom is p.

Example 6.4 Consider the following program Π_{10} :

$$p \leftarrow not\ a.$$

$$p \leftarrow not\ b.$$

$$a \leftarrow not\ b.$$

$$b \leftarrow not\ a.$$

Π_{10} has two stable models $\{p, a\}$ and $\{p, b\}$. The well-founded model of Π_{10} has empty sets of well-founded and unfounded atoms. Therefore, p is a consequence of Π_3 in the stable model semantics, while the answer to p in the well-founded semantics is *undefined*. □

Finally, let us look at the following example from [25]:

Example 6.5 Consider Π_{11} consisting of rules:

$$a \leftarrow not\ b.$$

$$b \leftarrow c, not\ a.$$

$$c \leftarrow a.$$

This program has one answer set $\{a, c\}$, and so has a and c as its consequences. The well-founded model of Π_{11} has no well-founded atoms. Its unfounded atoms are $\{a, b, c\}$ and hence, according to the well-founded semantics, atoms a, b, and c are undefined.

There are large classes of programs for which both, well-founded and stable model, semantics coincide. (See for instance [109], [13]). The SLDNF resolution is sound with respect to the well-founded semantics but it is not complete. Several attempts were made to define variants of SLDNF resolution which compute answers to goals according to the well-founded entailment. One interesting approach, SLS resolution, can be found in [108], [113]. SLS resolution is based on a type of an oracle and cannot therefore be viewed as an algorithm. There are however several algorithms and systems which can be viewed as SLS based approximations of the well-founded semantics [21], [15] One of the most powerful such systems, *XSB* (www.cs.sunysb.edu/ sbprolog/xsb-page.html) expands SLDNF with tabling and loop checking. Its use allows to avoid many of the loop related problems of Prolog. For instance, *XSB*'s answer to query $reachable(c)$ for program from Example 6.1 will be *no* (the Prolog interpreters will loop on this query).

7 Computational Complexity

In this section we will give a short overview of results on the computational complexity of A-Prolog programs.

One may wonder why we should be interested in analyzing the complexity of A-Prolog. This question has been addressed very clearly by Gottlob in [53], who pointed out that there are three main reasons, why one should be interested in studying the (worst case) complexity of logic programming formalisms. We report these reasons below.

First, the worst case complexity is a very good indicator of how many sources of structural complexity are inherent in a problem. For example, if a problem is NP-complete, then it contains basically one source of intractability, related to a choice to be made among exponentially many candidates. If a problem is Σ_2^P-complete then there are usually two intermingled sources of complexity. For instance, the problem of checking whether an A-Prolog program has an answer set is Σ_2^P-complete. The two sources of complexity are (1) the choice of a suitable interpretation which is a model of the reduct of the program and (2) the proof that this model is minimal.

Secondly, once the sources of complexity are identified, one can develop smart algorithms that take these sources into account. In addition, it becomes easier to discover tractable (i.e., polynomial) sub-cases by considering syntactic restrictions that eliminate all sources of intractability.

Finally, a precise complexity classification gives us valuable information about the algorithmic similarity and inter-translatability of different problems. For instance, both brave reasoning on normal (*or*-free) A-Prolog programs and brave reasoning on head-cycle free A-Prolog programs are NP-complete (see Table 2). Therefore, there are simple (i.e., polynomial) translations between these two reasoning tasks. In particular, this means that if one has implemented a reasoning engine (theorem prover) for one of these formalisms, this system can easily be adapted to become a reasoning engine for the other formalism. In most cases, the translations between two decision problems that are complete for the same complexity class can easily be deduced from the respective completeness proofs. At least, the underlying intuitions in these proofs may help to find a suitable translation scheme. On the other hand, if it is known that two problems are complete for different complexity classes in the polynomial hierarchy, the existence of a polynomial translation from the harder to the easier problem is unlikely. For example, brave reasoning with normal (non-disjunctive) A-Prolog programs is NP-complete (see Table 2). Therefore, unless the polynomial hierarchy collapses, a polynomial translation from full A-Prolog programs to normal programs cannot exist.

In summary, the complexity analysis of a problem gives us much more than merely a quantitative statement about its tractability or intractability in the worst case. Rather, locating a problem at the right level in the polynomial hierarchy gives us a deep *qualitative* knowledge about this problem. Moreover, the complexity analysis is a precious tools to develop efficient implementations of A-Prolog systems. For instance, consider the problem of checking whether a given set of literals is an answer set or not. This problem is co-NP-complete for general A-Prolog programs; while it is polynomial for head-cycle free programs (see Table 1). Consequently, a smart implementation of function *IsAnswerSet()* (see Figure 1) should be able to efficiently check this property if the program is head-cycle free. Indeed, in the A-Prolog system **DLV**, the function devoted to answer set checking recognizes whether the input program is head-cycle free or not. An efficient polynomial-time method is then applied on head-cycle free programs; while a backtracking procedure is employed on general (non head-cycle free) A-Prolog programs. Similar considerations apply also to other syntactic fragments having lower complexity than the general case. For instance, reasoning on normal (disjunction free) stratified programs should be performed by a polynomial time procedure (see Table 3) by an efficient A-Prolog system.

7.2 Preliminaries on Complexity: The Polynomial Hierarchy

We assume that the reader has some aquaintance with the concepts of NP-completeness and complexity theory, the book [103] is an excellent source for deepening the knowledge in this field.

The classes Σ_k^P, and Π_k^P of the *Polynomial Hierarchy (PH)* (cf. [119]) are defined as follows:

$$\Sigma_0^P = \Pi_0^P = \mathrm{P} \text{ and for all } k \geq 1, \Sigma_k^P = \mathrm{NP}^{\Sigma_{k-1}^P}, \Pi_k^P = \text{co-}\Sigma_k^P.$$

In particular, NP $= \Sigma_1^P$, and co-NP $= \Pi_1^P$. NP^C denotes the class of problems that are solvable in polynomial time on a nondeterministic Turing machine with an oracle for any problem π in the class C.

The oracle replies to a query in unit time, and thus, roughly speaking, models a call to a subroutine for π that is evaluated in unit time. If C has complete problems, then instances of any problem π' in C can be solved in polynomial time using an oracle for any C-complete problem π, by transforming them into instances of π; we refer to this by stating that an oracle for C is used. Notice that all classes C considered here have complete problems.

Observe that for all $k \geq 1$,

$$\Sigma_k^P \subseteq \Sigma_{k+1}^P \subseteq \text{PSPACE}; \qquad \text{and} \qquad \Pi_k^P \subseteq \Pi_{k+1}^P \subseteq \text{PSPACE};$$

each inclusion is widely conjectured to be strict. Note that, by the rightmost inclusions, all these classes contain only problems that are solvable in polynomial space. They allow, however, a finer grained distinction between NP-hard problems that are in PSPACE.

7.3 Main Problems Considered

We study the complexity of the following three important decision problems arising in A-Prolog:

Answer Set Checking. Given an A-Prolog program Π, and a set M of ground literals as input, decide whether M is an answer set of Π.
Brave reasoning. Given an A-Prolog program Π, and a ground literal l, decide whether l is true in some answer sets of Π (i.e., $\Pi \models_b l$).
Cautious reasoning. Given an A-Prolog program Π, and a ground literal l, decide whether l is true in all answer sets of Π (i.e., $\Pi \models l$).

7.4 Complexity Results

We analyze the computational complexity of the three decision problems mentioned above for ground (i.e., propositional) A-Prolog programs. An interesting issue is the impact of syntactical restrictions on the logic program Π. In particular, comparing the power of disjunction with the power of negation is intriguing [29].

Starting from normal positive programs (without negation and disjunction), we consider the effect of allowing the (combined) use of the following constructs:

- strong negation
- stratified (nonmonotonic) negation
- arbitrary negation
- head-cycle free disjunction
- arbitrary disjunction (*or*)

The complexity results for Answer Set Checking, Brave Reasoning and Cautious Reasoning over A-Prolog programs are summarized in Table 1, Table 2, and Table 3, respectively. Therein, each column refers to a specific form of negation, namely: $\{\}$ = no negation, $\neg$ = strong negation, not_s = stratified negation, not = unrestricted (possibly unstratified) negation. The lines of the tables specify the allowance of disjunction; in particular, $\{\}$ = no disjunction, or_h = head-cycle free disjunction, or = unrestricted (possibly not

	$\{\}$	$\{\neg\}$	$\{\textbf{not}_s\}$	$\{\neg, \textbf{not}_s\}$	$\{\textbf{not}\}$	$\{\neg, \textbf{not}\}$
$\{\}$	P	P	P	P	P	P
$\{or_h\}$	P	P	P	P	P	P
$\{or\}$	co-NP	co-NP	co-NP	co-NP	co-NP	co-NP

Table 1
The Complexity of Answer Set Checking in Syntactic Fragments of A-Prolog

	$\{\}$	$\{\neg\}$	$\{\textbf{not}_s\}$	$\{\neg, \textbf{not}_s\}$	$\{\textbf{not}\}$	$\{\neg, \textbf{not}\}$
$\{\}$	P	P	P	P	NP	NP
$\{or_h\}$	NP	NP	NP	NP	NP	NP
$\{or\}$	Σ_2^P	Σ_2^P	Σ_2^P	Σ_2^P	Σ_2^P	Σ_2^P

Table 2
The Complexity of Brave Reasoning in Syntactic Fragments of A-Prolog

head-cycle free) disjunction. Each entry of the table provides the complexity class of the corresponding fragment of the language. For an instance, the entry $(\{or_h\}, \{not_s\})$ defines the fragment of A-Prolog allowing head-cycle free disjunction and stratified negation. The corresponding entry in Table 2, namely NP, expresses that brave reasoning for this fragment is NP-complete. The results reported in the tables represent completeness under logspace reductions, they are taken from [29,53,36,35].

As expected, the results for brave and cautious reasoning are symmetric in most cases, that is, whenever the complexity of a fragment is C under brave reasoning, its complexity is co-C under cautious reasoning (recall that co-P = P).

Strong negation does not affect at all the complexity of reasoning; each column containing strong negation is equal to the corresponding column without it. Limiting the forms of disjunction and nonmonotonic negation reduces the respective powers. For disjunction free programs, brave reasoning is polynomial on stratified negation, while it becomes NP-complete if we allow unrestricted (nonmonotonic) negation. Brave reasoning is NP-complete on head-cycle free

	$\{\}$	$\{\neg\}$	$\{\textbf{not}_s\}$	$\{\neg, \textbf{not}_s\}$	$\{\textbf{not}\}$	$\{\neg, \textbf{not}\}$
$\{\}$	P	P	P	P	co-NP	co-NP
$\{or_h\}$	co-NP	co-NP	co-NP	co-NP	co-NP	co-NP
$\{or\}$	co-NP [9]	co-NP [9]	Π_2^P	Π_2^P	Π_2^P	Π_2^P

Table 3
The Complexity of Cautious Reasoning in Syntactic Fragments of A-Prolog

programs even if no form of negation is allowed. The complexity jumps one level higher in the polynomial hierarchy, up to Σ_2^P-complexity, if full disjunction is allowed. Thus, disjunction seems to be harder than negation, since the full complexity is reached already on positive programs, even without any kind of negation.

The picture is a bit different for cautious reasoning. Full disjunction alone is not sufficient to get the full complexity of cautious reasoning on A-Prolog (Π_2^P), which remains in co-NP if default negation is disallowed. Intuitively, to disprove that a literal l is a cautious conference of a program Π, it is sufficient to find a *model* (which does not need to be an answer set or a minimal model) which does not contain l. Indeed, for *not*-free programs, the existence of such a model guarantees that there exists also an answer set of Π which does not contain l. Therefore, under cautious inference, positive programs are easier to evaluate than programs with default negation; while, this is not true under brave inference modality.

The complexity results for Answer Set Checking, reported in Table 1, help us to understand the complexity of reasoning. Whenever Answer Set Checking is co-NP-complete for a fragment F, then the complexity of brave reasoning jumps up to the second level of the polynomial hierarchy (to Σ_2^P). Indeed, brave reasoning on full A-Prolog suffers of two "orthogonal" sources of complexity: (i) the exponential number of answer set "candidates", and (ii) the difficulty of checking whether a candidate M is an answer set (the minimality of M can be disproved by an exponential number of subsets of M). Now, disjunction (unrestricted or even head-cycle free) or unrestricted negation preserve the existence of source (i); while source (ii) exists only if full disjunction is allowed (see Table 1). As a consequence, reasoning lies at the second level of the polynomial hierarchy (Σ_2^P) for the A-Prolog fragments where both such complexity sources are present; while, it goes down to the first level of PH if only source (i) is present (unrestricted negation, or head-cycle free disjunction), falling down to level zero (P) if both sources are eliminated.

8 Further Program's Properties

In previous sections we already discussed some properties of logic programs, such as syntactic conditions guaranteeing existence and uniqueness of answer sets, the relationship between entailment under different semantics, soundness

[9] Note that here we consider the complexity of deciding if $\Pi \models L$, where L is either an atom or an atom negated by strong negation. Deciding if $\Pi \models not\ L$, i.e., if there is an answer set which does not contain L, is harder, precisely, this problem is Π_2^P-complete [35].

and completeness properties of various inference systems and algorithms, etc. In this section we'd like to briefly comment on some results establishing general properties of logic programming entailment relations.

Let us consider an A-Prolog program Π_{11} from example 6.5. Its answer set is $\{a, c\}$ and hence both a and c are the consequences of Π_{11}. When augmented with the fact c the program gains a second answer set $\{b, c\}$, and loses consequence a. The example demonstrates that the answer set entailment relation does not satisfy property:

$$\frac{\Pi \models a, \quad \Pi \models b}{\Pi \cup \{a.\} \models b} \tag{34}$$

called *cautious monotonicity.* The absence of cautious monotonicity is an unpleasant property of the answer set entailment. Among other things it prohibits the development of general inference algorithms for A-Prolog in which already proven lemmas are simply added to the program. We will say that a class of programs is cautiously monotonic if programs from this class satisfy condition 34. To give an example of such a class let us consider a syntactic condition on programs known as *order-consistency* [114].

Definition 8.1 For any *nlp* Π and atom a, Π_a^+ and Π_a^- are the smallest sets of atoms such that $a \in \Pi_a^+$ and, for every rule $r \in \Pi$,

- if $head(r) \in \Pi_a^+$ then $pos(r) \subseteq \Pi_a^+$ and $neg(r) \subseteq \Pi_a^-$,
- if $head(r) \in \Pi_a^-$ then $pos(r) \subseteq \Pi_a^-$ and $neg(r) \subseteq \Pi_a^+$,

Intuitively, Π_a^+ is the set of atoms on which atom a depends positively in Π, and Π_a^- is the set of atoms on which atom a depends negatively on Π. A program Π is *order-consistent* if there is a level mapping $||\ ||$ such that $||b|| < ||a||$ whenever $b \in \Pi_a^+ \cap \Pi_a^-$. That is, if a depends both positively and negatively on b, then b is mapped to a lower stratum. It is easy to see that program Π_3 from example 6.4 is order-consistent, while program Π_4 from example 6.5 is not. The following important theorem is due to H. Turner [123]

Theorem 8.1 If Π is an order-consistent program and atom a belongs to every answer set of Π, then every answer set of $\Pi \cup \{a.\}$ is an answer set of Π.

This immediately implies condition 34 for order-consistent programs.

A much simpler observation guarantees that all *nlp*'s under the answer set semantics have so called *cut* property: If an atom a belongs to an answer set X of Π, then X is an answer set of $\Pi \cup \{a.\}$.

Both results used together imply another nice property, called *cummulativity*: augmenting a program with one of its consequences does not alter its consequences. More precisely,

Theorem 8.2 If an atom a belongs to every answer set of an order-consistent program Π, then Π and $\Pi \cup \{a.\}$ have the same answer sets.

Semantic properties such as cummulativity, cut, and cautious monotonicity were originally formulated for analysis of nonmonotonic consequence relations ([46], [65]). Makinson's [75] handbook article includes a survey of such properties for nonmonotonic logics used in AI.

9 Conclusion

In this paper we described several important themes related to research on logic programming and knowledge representation in A-Prolog. The papers in this volume expand on these foundations. They are selected from those presented at *LPNMR99* – 5th International Conference on Logic Programming and Non-monotonic Reasoning, held in 1999 in El Paso, Texas. These papers are significant extensions of the respective versions presented at the conference.

Paper [72] illustrates the application of the declarative programming methodology of A-Prolog to the planning domain. The work in [54] addresses some important fixed-parameter complexity questions in artificial intelligence and nonmonotonic reasoning. Article [2] deals with the issue of updates in logic programming, introducing a language, called LUPS, for specifying dynamic changes in knowledge bases. Paper [31] reports on the successful application of the paradigm preference logic grammars to the problem of data standardization. Paper [76] defines an "annotated" version of the revision programs of Marek and Truszczynski. While revision programs are used to update, essentially, classical propositional interpretations (complete databases), annotated revision programs are more powerful allowing one to update the general "T-valuations". Finally, paper [118] describes an interesting linguistic extension of A-Prolog, which allows us to express cardinality constraints and weight constraints more naturally; it also illustrates one of the most popular A-Prolog systems, SMODELS.

There is a number of other logical languages and reasoning methods which can be viewed as alternatives to A-Prolog. They were developed in approximately the same time frame as A-Prolog, share the same roots and a number of basic ideas. The relationship and mutual fertilization between these approaches is a fascinated subject which was not addressed here. For more information the interested reader can look at [1,16,57,21,24,127].

Acknowledgement

The authors are grateful to Georg Gottlob for his contribution on complexity issues.

References

[1] J. Alferes and L. Pereira. *Reasoning With Logic Programming.* Springer Verlag, 1996.

[2] J. Alferes, L. Pereira, H. Przymusinska, T. Przymusinski LUPS - a language for updating logic programs *Artificial Intelligence*, Elsevier, this issue, 2002.

[3] K. Apt From Logic Programming to Prolog. C.A.R. Hoar series, Prentice Hall, 1997

[4] 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.

[5] K. Apt and D. Pedreschi. Proving termination in general prolog programs. In *Proc. of the Int'l Conf. on Theoretical Aspects of Computer Software (LNCS 526)*, pages 265–289. Springer Verlag, 1991.

[6] K. Apt and A. Pellegrini. On the occur-check free logic programs. *ACM Transaction on Programming Languages and Systems*, 16(3):687–726, 1994.

[7] K. Apt and R.Bol Logic Programming and negation: a survey. *Journal of Logic Programming*, 19,20 : 9–71,1994.

[8] C. Baral Knowledge representation, reasoning and declarative problem solving with answer sets. *Unpublished manuscript*, www.public.asu.edu/ cbaral/bahi/

[9] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19,20:73–148, 1994.

[10] C. Baral and M. Gelfond. Reasoning agents in dynamic domains. In J Minker, editor, *Logic Based AI.* Kluwer, pages 257–279, 2000

[11] R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. Annals of Mathematics and Artificial Intelligence, vol 12, pp. 53-87, 1994

[12] 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.

[13] N. Bidoit and C. Froidevaux. Negation by default and unstratifiable logic programs. *Theoretical Computer Science*, 79(1):86–112, 1991.

[14] Yu. Babovich, E. Erdem and V. Lifschitz. Fages' theorem and answer set programming. Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, 2000.

[15] R. Bol and L. Degerstedt. Tabulated resolution for the well-founded semantics. *Journal of Logic Programming*, 34(2):67–110, 1998.

[16] A. Bondarenko, P.M. Dung, R. Kowalski, F. Toni, An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence* 93(1-2) pages 63-101, 1997.

[17] G. Brewka, J. Dix, K. Konolige Nonmonotonic Reasoning: An Overview. *Stanford:CSLI Publications*, 1997.

[18] M. Cadoli, T. Eiter, and G. Gottlob. Default logic as a query language. IEEE Transactions on Knowledge and Data Engineering, 9(3), pages 448–463.

[19] F. Calimeri, W. Faber, N. Leone, and G. Pfeifer. Pruning Operators for Answer Set Programming Systems. DBAI-TR-01-10, Institut für Informationssysteme, Technische Universität Wien, Austria, April, 2001.

[20] J. Chen. Minimal knowledge + negation as failure = only knowing (sometimes). In *Proceedings of the Second Int'l Workshop on Logic Programming and Non-monotonic Reasoning, Lisbon*, pages 132–150, 1993.

[21] W. Chen, T. Swift, and D. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–201, 1995.

[22] P. Cholewinski, W. Marek, and M. Truszczyński. Default Reasoning System DeReS. In *Int'l Conf. on Principles of Knowledge Representation and Reasoning*, 518-528. Morgan Kauffman, 1996

[23] 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.

[24] D. De Schreye, M. Bruynooghe, B. Demoen, M. Denecker, G. Janssens, B. Martens. Project Report on LP+: A Second Generation Logic Programming Language. *AI Communications*, 13(1): 13-18, 2000.

[25] 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.

[26] W. Drabent. Completeness of SLDNF-resolution for non-floundering queries. *The Journal of Logic Programming*, 27(2):89-106, 1996.

[27] T. Eiter, N. Leone, C. Mateis., G. Pfeifer and F. Scarcello. A deductive system for nonmonotonic reasoning, *Proceedings of the 4rd Logic Programming and Non-Monotonic Reasoning Conference - LPNMR '97*, LNAI 1265, Springer-Verlag, Dagstuhl, Germania, Luglio 1997, pp. 363–374.

[28] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative problem solving using the **DLV** system. In J Minker, editor, *Logic Based AI*. Kluwer Academic Publishers, 2000, pp. 79–103.

[29] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

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

[31] B. Cui, T. Swift. Preference Logic Grammars: Semantics, Standardization, and Application to Data Standardization. *Artificial Intelligence*, Elsevier, this issue, 2002.

[32] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. *Lecture Notes in Artificial Intelligence - Recent Advances in AI Planning, Proc. of the 4th European Conference on Planning, ECP'97*, 1348:169–181, 1997

[33] D. East and M. Truszczyński. dcs: An implementation of DATALOG with Constraints. *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, 9–11 April, 2000.

[34] U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00)*, July 30 – August 3, 2000, Austin, Texas USA, pp. 417–422. AAAI Press / MIT Press, 2000.

[35] T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4), pp. 289–323, 1995.

[36] T. Eiter, N. Leone and D. Saccá. Expressive Power and Complexity of Partial Models for Disjunctive Deductive Databases. *Theoretical Computer Science*, Elsevier, 206(1–2), 1998, pp. 181–218.

[37] C. Elkan. A rational reconstruction of non-monotonic truth maintenance systems. *Artificial Intelligence*, 43, 1990.

[38] K. Esghi. Computing Stable Models by Using the ATMS. In *Proc. AAAI-90*, pages 272–277, 1990.

[39] E. Erdem, V. Lifschitz, and M. Wong. Wire routing and satisfiability planning. *Proc. of CL-2000*, pp. 822-836, 2000.

[40] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*. Japan, 3–5, September, pp. 135–139, 1999.

[41] W. Faber, N. Leone, G. Pfeifer. Pushing Goal Derivation in DLP Computations. *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, LNAIi 1730, Springer-Verlag, El Paso, Texas, 2–4 December, pp. 177–191, 1999.

[42] W. Faber, N. Leone, G. Pfeifer. Experimenting with Heuristics for Answer Set Programming. *Proceedings of the 17th International Joint Conference on Artificial Intelligence – IJCAI '01*, Morgan Kaufmann Publishers, Seattle, WA, USA, Agosto 2001, pp. 635–640.

[43] F. Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1(1):51–60, 1994.

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

[45] E. Franconi, A. Laureti Palma, N.Leone, S. Perri, F.Scarcello. Census Data Repair: A Challenging Application of Disjunctive Logic Programming. *Proceedings of LPAR'01*, Springer-Verlag, Cuba, December 2001.

[46] 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.

[47] L. Giordano and A. Martelli. Generalized stable models, truth maintenance and conflict resolution. In D. Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh International*, pages 427–441. The MIT Press, 1990.

[48] M. Gelfond. On stratified autoepistemic theories. In *Proc. AAAI-87*, pages 207–211, 1987.

[49] 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.

[50] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–387, 1991.

[51] M. Gelfond and V. Lifschitz. Representing Actions and Change by Logic Programs. *Journal of Logic Programming*, 17:301–323.

[52] M. Gelfond and T. Son. Reasoning with prioritized defaults. In J. Dix, L. M. Pereira, T. Przymusinski, editors, *Lecture Notes in Artificial Intelligence*, 1471, pp 164-224, 1998.

[53] G. Gottlob. Complexity and Expressive Power of Disjunctive Logic Programming. *Proceedings of the International Logic Programming Symposium (ILPS '94).* MIT Press, Ithaca NY, pp. 23–42, 1994.

[54] G. Gottlob , F. Scarcello, M. Sideri. Fixed parameter complexity in AI and nonmonotonic reasoning. *Artificial Intelligence*, Elsevier, this issue, 2002.

[55] 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.

[56] 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.

[57] A. C. Kakas, R. Kowalski, F. Toni, The Role of Abduction in Logic Programming, *Handbook of Logic in Artificial Intelligence and Logic Programming 5*, pages 235-324, D.M. Gabbay, C.J. Hogger and J.A. Robinson eds., Oxford University Press (1998).

[58] M. Kaminski. A note on the stable model semantics of logic programs. *Artificial Intelligence*, 96(2):467–479, 1997.

[59] M. Kaminski. A comparative study of open default theories. *Artificial Intelligence*, 77 (2): 285–319, 1995.

[60] C. Koch, N. Leone. Stable Model Checking Made Easy. *Proceedings of the 16th International Joint Conference on Artificial Intelligence – IJCAI'99*, Morgan Kaufmann Publishers, pp. 70–75, Sweden, August, 1999.

[61] R. Kowalski. Predicate logic as a programming language. *Information Processing 74*, pages 569–574, 1974.

[62] R. Kowalski. *Logic for Problem Solving*. North-Holland, 1979.

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

[64] K. Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7(3):231–245, 1989.

[65] D. Lehmann. What does a conditional knowledge base entail? In *Proceedings of KR 89*, pages 212–221, 1989.

[66] N. Leone, P. Rullo, F. Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, Academic Press, 135(2), 1997, pp. 69-112.

[67] V. Lifschitz. Closed-world data bases and circumscription. *Artificial Intelligence*, 27, 1985.

[68] 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.

[69] V. Lifschitz. On open defaults. In J. Lloyd, editor, *Computational Logic: Symposium Proceedings*, pages 80–95. Springer, 1990.

[70] 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.

[71] V. Lifschitz. Foundations of logic programming. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 69–128. CSLI Publications, 1996.

[72] V. Lifschitz. Answer Set Planning. *Artificial Intelligence*, Elsevier, this issue, 2002.

[73] V. Lifschitz. Circumscription, In D.M. Gabbay, C.J.Hogger, and J.A.Robinson, editors, *The Handbook on Logic in AI and Logic Programming*, volume 3, pp 298–352, Oxford University Press, 1994

[74] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of disjunctive logic programming.* The MIT Press, 1992.

[75] D. Makinson. General patterns in nonmonotonic reasoning. In D.M. Gabbay, C.J.Hogger, and J.A.Robinson, editors, *The Handbook on Logic in AI and Logic Programming*, volume 3, pp 35–110, Oxford University Press, 1993

[76] V. Marek, I. Pivkina, M. Truszczyński. Annotated revision programs *Artificial Intelligence*, Elsevier, this issue, 2002.

[77] 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.

[78] 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.

[79] W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM.*, 3(38):588–619, 1991.

[80] 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.

[81] W. Marek, and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, 375–398, Spring-Verlag. 1999.

[82] W. Marek, and M. Truszczyński. *Nonmonotonic Logic*, Springer-Verlag, Berlin, 1993.

[83] 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.

[84] 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.

[85] J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1, 2):27–39, 1980.

[86] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986.

[87] N. McCain and H. Turner. Satisfiability Planning with Causal Theories. *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pp. 212–223. Morgan Kaufmann Publishers, 1998.

[88] D. McDermott. Nonmonotonic logic II: Nonmonotonic modal theories. *Journal of the ACM.*, 29(1):33–57, 1982.

[89] D. McDermott and J. Doyle. Nonmonotonic logic I. *Artificial Intelligence*, 13(1,2):41–72, 1980.

[90] J. Minker. On indefinite data bases and the closed world assumption. In *Proc. of CADE-82*, pages 292–308, 1982.

[91] J. Minker Overview of disjunctive logic programming. *Annals of mathematics and artificial Intelligence*, 12:1–24, 1994.

[92] J. Minker Logic and Databases: a 20 year retrospective. In Levesque,H. and Pirri. F,editors. Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter, pages 234–299, Springer, 1999.

[93] J. Minker, editor. Logic-Based Artificial Intelligence. Kluwer Academic Publishers, 2000.

[94] M. Minsky A framework for representing knowledge. In Winston P. editor, *The Psych. of computer vision*, pages 211–277, McGraw-Hill, NY.

[95] R. Moore. Semantical Considerations on Nonmonotonic Logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[96] D. Nelson. Constructible falsity. *Journal of Symbolic Logic*, 14:16–26, 1949.

[97] A. Nerode and R. Shore Logic for Applications. Springer, 1997.

[98] Niemela, I. Logic programs with stable model semantics as a constraint programming paradigm. In *Annals of Mathematics and Artificial Intelligence*, 25(3-4), 241–273, 1999.

[99] I. Niemela and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proc. 4th international conference on Logic programming and non-monotonic reasoning*, pages 420–429, 1997.

[100] T. Soininen and I. Niemela. Developing a declarative rule language for applications in program configuration. *In practical aspects of declarative languages*, LNCS 1551, pages 305-319, 1999.

[101] U. Nilsson and J. Maluszynski. *Logic, Programming and Prolog*, www.ida.liu.se/ ulfni/lpp

[102] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. A-Prolog decision support system for the Space Shuttle. *In Proc. of Third International Symposium on Practical Aspects of Declarative Languages*, LNCS 1990, pages 169–183, 2001.

[103] C.H. Papadimitriou. *Computational Complexity*, Addison-Wesley, 1994.

[104] 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.

[105] D. Pearce. A new logical characterization of stable models and answer sets. In J. Dix,L. Pereira, and T. Przymusinski, editors. *Non-Monotonic Extensions of Logic Programming, (Lecture Notes in Artificial Intelligence 1216), pages 57–70, Springer-Verlag, 1997.*

[106] D. Pearce. From here to there: Stable negation in logic programming. In D. Gabbay and H. Wansing, editors. *What is negation?*, Kluwer, 1999.

[107] 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.

[108] T. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. *In Proc. of the 8th Symposium on Principles of Database Systems*, pages 11–21, 1989.

[109] 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.

[110] 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.

[111] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1,2):81–132, 1980.

[112] R. Reiter. Circumscription implies predicate completion (sometimes). In *Proc. of IJCAI-82*, pages 418–420, 1982.

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

[114] T. Sato. Completed logic programs and their consistency. *Journal of Logic Programming*, 9 : 33–44, 1990

[115] C. Sakama and K. Inoue Prioritized Logic Programming and its Application to Commonsense Reasoning, *Artificial Intelligence* 123(1-2):185-222, Elsevier, 2000.

[116] M. Shanahan. *Solving the frame problem: A mathematical investigation of the commonsense law of inertia.* MIT press, 1997.

[117] G. Schwarz. Autoepistemic logic of knowledge. In Anil Nerode, Victor Marek, and Subrahmanian V. S., editors, *Logic Programming and Non–monotonic Reasoning: Proc. of the First Int'l Workshop*, pages 260–274, 1991.

[118] P. Simons, I. Niemela, T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, Elsevier, this issue, 2002.

[119] L. Stockmeyer. Classifying the Computational Complexity of Problems. *Journal of Symbolic Logic*, **52**(1), 1–43, 1987.

[120] K. Stroetman. A Completeness Result for SLDNF-Resolution. *Journal of Logic Programming*, 15:337–355, 1993.

[121] 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.

[122] H. Turner. Representing actions in logic programs and default theories. *Journal of Logic Programming*, 31(1-3):245–298, May 1997.

[123] H. Turner Order-consistent programs are cautiously monotonic. Theory and Practice of Logic Programming (to appear)

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

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

[126] G. Wagner. Logic programming with strong negation and inexact predicates. *Journal of Logic and Computation*, 1(6):835–861, 1991.

[127] J.-H. You and L. Yuan. A Three-Valued Semantics for Deductive Databases and Logic Programs. *Journal of Computer and System Sciences*, 49:334–361, 1994.