Handbook of Knowledge Representation
Edited by F. van Harmelen, V. Lifschitz and B. Porter
© 2008 Elsevier B.V. All rights reserved
DOI: 10.1016/S1574-6526(07)03007-6

Chapter 7

Answer Sets

Michael Gelfond

7.1 Introduction

This chapter is an introduction to Answer Set Prolog—a language for knowledge representation and reasoning based on the *answer set/stable model* semantics of logic programs [44, 45]. The language has roots in declarative programing [52, 65], the syntax and semantics of standard Prolog [24, 23], disjunctive databases [66, 67] and nonmonotonic logic [79, 68, 61]. Unlike "standard" Prolog it allows us to express disjunction and "classical" or "strong" negation. It differs from many other knowledge representation languages by its ability to represent *defaults*, i.e., statements of the form "*Elements of a class C normally satisfy property P*". A person may learn early in life that parents normally love their children. So knowing that Mary is John's mother he may conclude that Mary loves John and act accordingly. Later he may learn that Mary is an exception to the above default, conclude that Mary does not really like John, and use this new knowledge to change his behavior. One can argue that a substantial part of our education consists in learning various defaults, exceptions to these defaults, and the ways of using this information to draw reasonable conclusions about the world and the consequences of our actions. Answer Set Prolog provides a powerful logical model of this process. Its syntax allows for the simple representation of defaults and their exceptions, its consequence relation characterizes the corresponding set of valid conclusions, and its inference mechanisms often allow a program to find these conclusions in a reasonable amount of time.

There are other important types of statements which can be nicely expressed in Answer Set Prolog. This includes the causal effects of actions ("statement F becomes true as a result of performing an action a"), statements expressing a lack of information ("it is not known if statement P is true or false"), various completeness assumptions "statements not entailed by the knowledge base are false", etc.

There is by now a comparatively large number of inference engines associated with Answer Set Prolog. SLDNF-resolution based goal-oriented methods of "classical" Prolog and its variants [22] are sound with respect to the answer set semantics of their programs. The same is true for fix-point computations of deductive databases [93]. These systems can be used for answering various queries to a subset of

Answer Set Prolog which does not allow disjunction, "classical" negation, and rules with empty heads. In the last decade we have witnessed the coming of age of inference engines aimed at computing the answer sets of Answer Set Prolog programs [71, 72, 54, 29, 39, 47]. These engines are often referred to as *answer set solvers*. Normally they start with grounding the program, i.e., instantiating its variables by ground terms. The resulting program has the same answer sets as the original but is essentially propositional. The grounding techniques implemented by answer set solvers are rather sophisticated. Among other things they utilize algorithms from deductive databases, and require a good understanding of the relationship between various semantics of logic programming. The answer sets of the grounded program are often computed using substantially modified and expanded satisfiability checking algorithms. Another approach reduces the computation of answer sets to (possibly multiple) calls to satisfiability solvers [3, 47, 58].

The method of solving various combinatorial problems by reducing them to finding the answer sets of Answer Set Prolog programs which declaratively describe the problems is often called the *answer set programming paradigm* (*ASP*) [70, 62]. It has been used for finding solutions to a variety of programming tasks, ranging from building decision support systems for the Space Shuttle [74] and program configuration [87], to solving problems arising in bio-informatics [9], zoology and linguistics [20]. On the negative side, Answer Set Prolog in its current form is not adequate for reasoning with complex logical formulas—the things that classical logic is good at—and for reasoning with real numbers.

There is a substantial number of natural and mathematically elegant extensions of the original Answer Set Prolog. A long standing problem of expanding answer set programming by aggregates—functions on sets—is approaching its final solution in [33, 32, 88, 76, 35]. The rules of the language are generalized [38] to allow nested logical connectives and various means to express preferences between answer sets [18, 25, 82]. Weak constraints and consistency restoring rules are introduced to deal with possible inconsistencies [21, 7]. The logical reasoning of Answer Set Prolog is combined with probabilistic reasoning in [14] and with qualitative optimization in [19]. All of these languages have at least experimental implementations and an emerging theory and methodology of use.

7.2 Syntax and Semantics of Answer Set Prolog

We start with a description of syntax and semantics of Answer Set Prolog—a logic programming language based on the answer sets semantics of [45]. In what follows we use a standard notion of a sorted signature from classical logic. We will assume that our signatures contain sort N of non-negative integers and the standard functions and relations of arithmetic. (Nothing prevents us from allowing other numerical types but doing that will lengthen some of our definitions. So N will be the only numerical sort discussed in this paper.) Terms and atoms are defined as usual. An atom $p(\bar{t})$ and its negation $\neg p(\bar{t})$ will be referred to as *literals*. Literals of the form $p(\bar{t})$ and $\neg p(\bar{t})$ are called *contrary*. A rule of Answer Set Prolog is an expression of the form

$$l_0 \; or \; \dots \; or \; l_k \leftarrow l_{k+1}, \dots, l_m, not\; l_{m+1}, \dots, not\; l_n, \tag{7.1}$$

where l_i's are literals. Connectives *not* and *or* are called *negation as failure* or *default negation*, and *epistemic disjunction*, respectively. Literals possibly preceded by default negation are called *extended literals*.

A rule of Answer Set Prolog which has a nonempty head and contains no occurrences of $\neg$ and no occurrences of *or* is called an *nlp* rule. Programs consisting of such rules will be referred to as *nlp* (*normal logic program*).

If r is a rule of type (7.1) then $head(r) = \{l_0, \dots, l_k\}$, $pos(r) = \{l_{k+1}, \dots, l_m\}$, $neg(r) = \{l_{m+1}, \dots, l_n\}$, and $body(r) = \{l_{k+1}, \dots, l_m, not\ l_{m+1}, \dots, not\ l_n\}$. If $head(r) = \emptyset$ rule r is called a *constraint* and is written as

$$\leftarrow l_{k+1}, \dots, l_m, not\ l_{m+1}, \dots, not\ l_n. \tag{7.2}$$

If $k = 0$ then we write

$$l_0 \leftarrow l_1, \dots, l_m, not\ l_{m+1}, \dots, not\ l_n. \tag{7.3}$$

A rule r such that $body(r) = \emptyset$ is called a *fact* and is written as

$$l_0\ or\ \dots\ or\ l_k. \tag{7.4}$$

Rules of Answer Set Prolog will often be referred to as *logic programming rules*.

Definition 7.2.1. *A* program *of Answer Set Prolog is a pair* $\{\sigma, \Pi\}$ *where* σ *is a signature and* Π *is a collection of logic programming rules over* σ.

In what follows we adhere to the convention used by all the inference engines of Answer Set Prolog and end every rule by a ".".

Consider for instance a signature σ with two sorts, $\tau_1 = \{a, b\}$ and $\tau_2 = N$. Suppose that σ contains predicate symbols $p(\tau_1)$, $q(\tau_1, \tau_2)$, $r(\tau_1)$, and the standard relation $<$ on N. The signature, together with rules

$$\Pi_0 \begin{cases} q(a, 1). \\ q(b, 2). \\ p(X) \leftarrow K + 1 < 2, \\ \qquad\qquad q(X, K). \\ r(X) \leftarrow not\ p(X). \end{cases}$$

constitute a program of Answer Set Prolog. Capital letters X and K denote variables of the appropriate types.

In this paper we will often refer to programs of Answer Set Prolog as *logic programs* and denote them by their second element Π. The corresponding signature will be denoted by $\sigma(\Pi)$. If $\sigma(\Pi)$ is not given explicitly, we assume that it consists of symbols occurring in the program.

To give the semantics of Answer Set Prolog we will need the following terminology. Terms, literals, and rules of program Π with signature σ are called *ground* if they contain no variables and no symbols for arithmetic functions. A program is called *ground* if all its rules are ground. A rule r' is called a *ground instance* of a rule r of Π if it is obtained from r by:

1. replacing r's non-integer variables by properly typed ground terms of $\sigma(\Pi)$;
2. replacing r's variables for non-negative integers by numbers from N;
3. replacing the remaining occurrences of numerical terms by their values.

A program $gr(\Pi)$ consisting of all ground instances of all rules of Π is called the *ground instantiation* of Π. Obviously $gr(\Pi)$ is a ground program.

Below is the ground instantiation of program Π_0

$$gr(\Pi_0)\begin{cases} q(a,1). \\ q(b,2). \\ p(a) \quad \leftarrow 1 < 2, \\ \qquad\qquad q(a,0). \\ p(a) \quad \leftarrow 2 < 2, \\ \qquad\qquad q(a,1). \\ \qquad\quad \ldots \\ r(a) \quad \leftarrow \mathit{not}\ p(a). \\ r(b) \quad \leftarrow \mathit{not}\ p(b). \end{cases}$$

Consistent sets of ground literals over σ, containing all arithmetic literals which are true under the standard interpretation of their symbols, are called *partial interpretations* of σ. Let l be a ground literal. By $\bar{l}$ we denote the literal contrary to l. We say that l is *true* in a partial interpretation S if $l \in S$; l is *false* in S if $\bar{l} \in S$; otherwise l is *unknown* in S. An extended literal *not* l is *true* in S if $l \notin S$; otherwise it is *false* in S. A set U of extended literals is understood as conjunction, and sometimes will be written with symbol $\wedge$. U is *true* in S if all elements of U are true in S; U is *false* in S if at least one element of U is false in S; otherwise U is *unknown*. Disjunction D of literals is *true* in S if at least one of its members is true in S; D is *false* in S if all members of D are false in S; otherwise D is *unknown*. Let e be an extended literal, a set of extended literals, or a disjunction of literals. We refer to such expressions as *formulas* of σ. For simplicity we identify expressions $\neg(l_1\ \mathit{or}\ \ldots\ \mathit{or}\ l_n)$ and $\neg(l_1,\ldots,l_n)$ with the conjunction $\bar{l}_1 \wedge \cdots \wedge \bar{l}_n$. and disjunction $\bar{l}_1\ \mathit{or}\ \ldots\ \mathit{or}\ \bar{l}_n$, respectively. We say that *S satisfies e* if e is true in S. *S satisfies a logic programming rule r* if S satisfies r's head or does not satisfy its body.

Our definition of semantics of Answer Set Prolog will be given for ground programs. Rules with variables will be used only as a shorthand for the sets of their ground instances. This approach is justified for the so called closed domains, i.e. domains satisfying the *domain closure assumption* [78] which asserts that *all objects in the domain of discourse have names in the signature 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 Answer Set Prolog for open domains can be found in [11, 84, 49].

The answer set semantics of a logic program Π assigns to Π a collection of *answer sets*—partial interpretations of $\sigma(\Pi)$ corresponding to possible sets of beliefs which can be built by a rational reasoner on the basis of rules of Π. In the construction of such a set, S, the reasoner is assumed to be guided by the following informal principles:

- S must satisfy the rules of Π;
- the reasoner should adhere to the *rationality principle* which says that *one shall not believe anything one is not forced to believe*.

The precise definition of answer sets will be first given for programs whose rules do not contain default negation. Let Π be such a program and let S be a partial interpretation of $\sigma(\Pi)$.

Definition 7.2.2 (Answer set—part one). *A partial interpretation S of $\sigma(\Pi)$ is an* answer set *for Π if S is minimal (in the sense of set-theoretic inclusion) among the partial interpretations satisfying the rules of Π.*

The rationality principle is captured in this definition by the minimality requirement.

Example 7.2.1 (Answer sets). A program

$$\Pi_1 \begin{cases} q(a). \\ p(a). \\ r(a) \leftarrow p(a), \\ \qquad\qquad q(a). \\ r(b) \leftarrow q(b). \end{cases}$$

has one answer set, $\{q(a), p(a), r(a)\}$.

A program

$$\Pi_2 = \{q(a) \textit{ or } q(b).\}$$

has two answer sets, $\{q(a)\}$ and $\{q(b)\}$, while a program

$$\Pi_3 \begin{cases} q(a) \textit{ or } q(b). \\ \neg q(a). \end{cases}$$

has one answer set $\{\neg q(a), q(b)\}$.

We use the symbol *or* instead of classical $\vee$ to stress the difference between the two connectives. A formula $A \vee B$ of classical logic says that "A is *true* or B is *true*" while a rule, A *or* B, may be interpreted epistemically and means that every possible set of reasoner's beliefs must satisfy A or satisfy B. To better understand this intuition consider the following examples.

Example 7.2.2 (More answer sets). It is easy to see that program

$$\Pi_4 \begin{cases} p(a) \leftarrow q(a). \\ p(a) \leftarrow \neg q(a). \end{cases}$$

has unique answer set $A = \emptyset$, while program

$$\Pi_5 \begin{cases} p(a) \leftarrow q(a). \\ p(a) \leftarrow \neg q(a). \\ q(a) \textit{ or } \neg q(a). \end{cases}$$

has two answer sets, $A_1 = \{q(a), p(a)\}$ and $A_2 = \{\neg q(a), p(a)\}$. The answer sets reflect the epistemic interpretation of *or*. The statement $q(a)$ *or* $\neg q(a)$ is not a tautology. The reasoner associated with program Π_4 has no reason to believe either $q(a)$ nor $\neg q(a)$. Hence he believes neither which leads to the answer set of Π_4 being empty. The last rule of program Π_5 forces the reasoner to only consider sets of beliefs which contain either $q(a)$ or $\neg q(a)$ which leads to two answer sets of Π_5.

Note also that it would be wrong to view epistemic disjunction *or* as exclusive. It is true that, due to the minimality condition in the definition of answer set, program

$$\Pi_2 = \{q(a) \text{ or } q(b).\}$$

has two answer sets, $A_1 = \{q(a)\}$ and $A_2 = \{q(b)\}$. But consider a query $Q = q(a) \wedge q(b)$. Since neither Q nor $\neg Q$ are satisfied by answer sets A_1 and A_2 the Π_5's answer to query Q will be *unknown*. The exclusive interpretation of *or* requires the definite negative answer. It is instructive to contrast Π_5 with a program

$$\Pi_6 = \Pi_2 \cup \{\neg q(a) \text{ or } \neg q(b)\}$$

which has answer sets $A_1 = \{q(a), \neg q(b)\}$ and $A_2 = \{q(b), \neg q(a)\}$ and clearly contains $\neg Q$ among its consequences.

The next two programs show that the connective $\leftarrow$ should not be confused with classical implication. Consider a program

$$\Pi_7 \begin{cases} \neg p(a) \leftarrow q(a). \\ q(a). \end{cases}$$

Obviously it has unique answer set $\{q(a), \neg p(a)\}$. But the program

$$\Pi_8 \begin{cases} \neg q(a) \leftarrow p(a). \\ q(a). \end{cases}$$

obtained from Π_7 by replacing its first rule by the rule's "contrapositive" has a different answer set, $\{q(a)\}$.

To extend the definition of answer sets to arbitrary programs, take any program Π, and let S be a partial interpretation of $\sigma(\Pi)$. The *reduct*, Π^S, of Π relative to S is the set of rules

$$l_0 \text{ or } \ldots \text{ or } l_k \leftarrow l_{k+1}, \ldots, l_m$$

for all rules (7.1) in Π such that $\{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$. Thus Π^S is a program without default negation.

Definition 7.2.3 (Answer set—part two). *A partial interpretation S of $\sigma(\Pi)$ is an answer set for Π if S is an answer set for Π^S.*

The relationship between this fix-point definition and the informal principles which form the basis for the notion of answer set is given by the following proposition.

Proposition 7.2.1. (*See Baral and Gelfond [11].*) *Let S be an answer set of logic program Π.*

(a) *S is closed under the rules of the ground instantiation of Π.*

(b) *If literal $l \in S$ then there is a rule r from the ground instantiation of Π such that the body of r is satisfied by S and l is the only literal in the head of r satisfied by S.*

Rule r from (b) "forces" the reasoner to believe l.

Definition 7.2.4 (Entailment). *A program Π* entails *a ground literal l ($\Pi \models l$) if l is satisfied by every answer set of Π.*

(Sometimes the above entailment is referred to as *cautious*.) Program Π representing knowledge about some domain can be queried by a user with a query q. For simplicity we assume that q is a ground formula of $\sigma(\Pi)$.

Definition 7.2.5 (Answer to a query). *We say that the program Π's answer to a query q is* yes *if $\Pi \models q$,* no *if $\Pi \models \neg q$, and* unknown *otherwise.*

Example 7.2.3. Consider for instance a logic program

$$\Pi_9 \begin{cases} p(a) \leftarrow not\ q(a). \\ p(b) \leftarrow not\ q(b). \\ q(a). \end{cases}$$

Let us first use the informal principles stated above to find an answer set, A, of Π_9. Since A must be closed under the rules of Π, it must contain $q(a)$. There is no rule forcing the reasoner to believe $q(b)$. This implies that $q(b) \notin A$. Finally, the second rule forces the reasoner to believe $p(b)$. The first rule is already satisfied and hence the construction is completed.

Using the definition of answer sets one can easily show that $A = \{q(a), p(b)\}$ 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_9. Thus $\Pi_9 \models q(a)$, $\Pi_9 \not\models q(b)$, $\Pi_9 \not\models \neg q(b)$ and Π_9's answers to queries $q(a)$ and $q(b)$ will be *yes* and *unknown*, respectively. If we expand Π_0 by a rule

$$\neg q(X) \leftarrow not\ q(X) \tag{7.5}$$

the resulting program

$$\Pi_{10} = \Pi_9 \cup (7.5)$$

would have the answer set $S = \{q(a), \neg q(b), p(b)\}$ and hence the answer to query $q(b)$ will become *no*.

The notion of answer set is an extension of an earlier notion of stable model defined in [44] for normal logic programs. 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 (*CWA*), [78] '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,

$$CWA(\Pi) = \Pi \cup \{\neg p(X_1, \ldots, X_n) \leftarrow not\ p(X_1, \ldots, X_n)\}$$

for every predicate symbol p of Π. Now it can be shown that for any ground literal l, $\Pi \models_s l$ iff $\Pi \models l$. Of course the closed world assumption does not have to be used for all of the relations of the program. If complete information is available about a particular relation p we call such relation *closed* and write $\neg p(X_1, \ldots, X_n) \leftarrow not\ p(X_1, \ldots, X_n)$. Relations which are not closed are referred to as *open*. Examples of open and closed relations will be given in Section 7.4.

7.3 Properties of Logic Programs

There is a large body of knowledge about mathematical properties of logic programs under the answer set semantics. The results presented in this section are aimed at providing a reader with a small sample of this knowledge. Due to the space limitations the presentation will be a mix of precise statements and informal explanations. For a much more complete coverage one may look at [8, 37].

7.3.1 Consistency of Logic Programs

Programs of Answer Set Prolog may have one, many, or zero answer sets. One can use the definition of answer sets to show that programs

$$\Pi_{11} = \{p(a) \leftarrow not\ p(a).\},$$
$$\Pi_{12} = \{p(a). \quad \neg p(a).\},$$

and

$$\Pi_{13} = \{p(a). \quad \leftarrow p(a).\}$$

have no answer sets while program

$$\Pi_{14} \begin{cases} e(0). \\ e(X+2) \leftarrow not\ e(X). \\ p(X+1) \leftarrow e(X), not\ p(X). \\ p(X) \quad\quad \leftarrow e(X), not\ p(X+1). \end{cases}$$

has an infinite collection of them. Each answer set of Π_{14} consists of atoms $\{e(0), e(3), e(4), e(7), e(8), \ldots\}$ and a choice of $p(n)$ or $p(n+1)$ for each integer n satisfying e.

Definition 7.3.1. *A logic program is called* consistent *if it has an answer set.*

Inconsistency of a program can reflect an absence of a solution to the problem it models. It can also be caused by the improper use of the connective $\neg$ and/or constraints as in programs Π_{12} and Π_{13} or by the more subtly incorrect use of default negation as in Π_{11}. The simple transformation described below [42, 11] reduces programs of Answer Set Prolog to programs without $\neg$.

For any predicate symbol p occurring in Π, let p' be a new predicate symbol of the same arity. The atom $p'(\bar{t})$ will be called the *positive form* of the negative literal $\neg p(\bar{t})$. Every positive literal is, by definition, its own positive form. The positive form of a literal l will be denoted by l^+. Program Π^+, called *positive form* of Π, is obtained from Π by replacing each rule (7.1) by

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

and adding the rules

$$\leftarrow p(\bar{t}), p'(\bar{t})$$

for every atom $p(\bar{t})$ of $\sigma(\Pi)$. For any set S of literals, S^+ stands for the set of the positive forms of the elements of S.

Proposition 7.3.1. *A set S of literals of $\sigma(\Pi)$ is an answer set of Π if and only if S^+ is an answer set of Π^+.*

This leaves the responsibility for inconsistency to the use of constraints and default negation. It is of course important to be able to check consistency of a logic program. Unfortunately in general this problem is undecidable Of course consistency can be decided for programs with finite Herbrand universes but the problem is complex. Checking consistency of such a program is Σ_2^P [27]. For programs without epistemic disjunction and default negation checking consistency belongs to class P; if no epistemic disjunction is allowed the problem is in NP [85]. It is therefore important to find conditions guaranteeing consistency of logic programs. In what follows we will give an example of such a condition.

Definition 7.3.2 (Level mapping). *Functions $\| \ \|$ from ground atoms of $\sigma(\Pi)$ to natural numbers*[1] *are called* level mappings *of Π.*

The level $\|D\|$ where D is a disjunction or a conjunction of literals is defined as the *minimum level of atoms occurring in literals from D'*. (Note that this implies that $\|l\| = \|\neg l\|$.)

Definition 7.3.3 (Stratification). *A logic program Π is called* locally stratified *if $gr(\Pi)$ does not contain occurrences of $\neg$ and there is a level mapping $\| \ \|$ of Π such that for every rule r of $gr(\Pi)$*

1. *For any $l \in pos(r)$, $\|l\| \leqslant \|head(r)\|$;*
2. *For any $l \in neg(r)$, $\|l\| < \|head(r)\|$.*

If, in addition, for any predicate symbol p, $\|p(\bar{t}_1)\| = \|p(\bar{t}_2)\|$ for any $\bar{t}_1$ and $\bar{t}_2$ the program is called stratified [1, 77].

[1] For simplicity we consider a special case of the more general original definition which allows arbitrary countable ordinals.

It is easy to see that among programs Π_0–Π_{14} only programs Π_0, Π_1, Π_2, and Π_9 are (locally) stratified.

Theorem 7.3.1 (Properties of locally stratified programs).

- *A locally stratified program is consistent.*
- *A locally stratified program without disjunction has exactly one answer set.*
- *The above conditions hold for a union of a locally stratified program and a collection of closed world assumptions, i.e., rules of the form*

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

  for some predicate symbols p.

The theorem immediately implies existence of answer sets of programs Π_9 and Π_{10} from the previous section.

We now use the notion of level mapping to define another syntactic condition on programs known as *order-consistency* [83].

Definition 7.3.4. *For any nlp Π and ground atom a, Π_a^+ and Π_a^- are the smallest sets of ground atoms such that $a \in \Pi_a^+$ and, for every rule $r \in gr(\Pi)$,*

- *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.

Obviously, every locally stratified *nlp* is order-consistent. The program

$$\Pi_{14} \begin{cases} p(X) \leftarrow not\ q(X). \\ q(X) \leftarrow not\ p(X). \\ r(X) \leftarrow p(X). \\ r(X) \leftarrow q(X). \end{cases}$$

with signature containing two object constants, c_1 and c_2 is order-consistent but not stratified, while the program

$$\Pi_{15} \begin{cases} a \leftarrow not\ b. \\ b \leftarrow c, \\ \qquad not\ a. \\ c \leftarrow a. \end{cases}$$

is not order-consistent.

Theorem 7.3.2 (First Fages' Theorem, [34]). *Order-consistent programs are consistent.*

7.3.2 Reasoning Methods for Answer Set Prolog

There are different algorithms which can be used for reasoning with programs of Answer Set Prolog. The choice of the algorithm normally depends on the form of the program and the type of queries one wants to be answered. Let us start with a simple example.

Definition 7.3.5 (Acyclic programs). *An nlp Π is called* acyclic [2] *if there is a level mapping $\| \ \|$ of Π such that for every rule r of $gr(\Pi)$ and every literal l which occurs in $pos(r)$ or $neg(r)$, $\|l\| < \|head(r)\|$.*

Obviously an acyclic logic program Π is stratified and therefore has a unique answer set. It can be shown that queries to an acyclic program Π can be answered by the SLDNF resolution based interpreter of Prolog. To justify this statement we will introduce the notion of Clark's completion [23]. The notion provided the original declarative semantics of *negation as finite failure* of the programming language Prolog. (Recall that in our terminology programs of Prolog are referred to as *nlp*.)

Let us consider the following three step transformation of a *nlp* Π into a collection of first-order formulae.

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

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

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

$$\alpha_1(\Pi) = \{\alpha_1(r) \colon r \in \Pi\}.$$

Step 2: For each predicate symbol 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

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

if $j \geqslant 1$ and by

$$\forall X_1 \ldots X_k \colon \neg p(X_1, \ldots, X_k)$$

if $j = 0$.

Step 3: Expand the resulting set of formulas by *free equality axioms*:

$$f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n) \supset X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n,$$

$$f(X_1, \ldots, X_n) = g(Y_1, \ldots, Y_n) \quad \text{for all } f \text{ and } g \text{ such that } f \neq g,$$

$X \neq t$ for each variable X and term t such that X is different from t and X occurs in t.

All the variables in free equality axioms are universally quantified; binary relation $=$ does not appear in Π; it is interpreted as identity in all models.

Definition 7.3.6 (Clark's completion, [23]). *The resulting first-order theory is called* Clark's completion *of Π and is denoted by $Comp(\Pi)$.*

Consider a program

$$\Pi_{16} \begin{cases} p(X) \leftarrow \mathit{not}\ q(X), \\ \qquad\qquad \mathit{not}\ r(X). \\ p(a). \\ q(b). \end{cases}$$

with the signature containing two object constants, a and b. It is easy to see that, after some simplification, $Comp(\Pi_{16})$ will be equivalent to the theory consisting of axioms

$$\begin{aligned} &\forall X\colon\ p(X) \equiv (\neg q(X) \vee X = a), \\ &\forall X\colon\ q(X) \equiv X = b, \\ &\forall X\colon\ \neg r(X) \end{aligned}$$

and the free equality axioms. One may also notice that the answer set $\{p(a), p(b), q(b)\}$ of Π_{16} coincides with the unique Herbrand model of $Comp(\Pi_{16})$.

The following theorem [1] generalizes this observation.

Theorem 7.3.3. *If Π is acyclic then the unique answer set of Π is the unique Herbrand model of Clark's completion of Π.*

The theorem is important since it allows us to use a large number of results about soundness and completeness of SLDNF resolution of Prolog with respect to Clark's semantics to guarantee these properties for acyclic programs with respect to the answer set semantics. Together with some results on termination this often guarantees that the SLDNF resolution based interpreter of Prolog will always terminate on atomic queries and produce the intended answers. Similar approximation of the Answer Set Prolog entailment for larger classes of programs with unique answer sets can be obtained by the system called *XSB* [22] implementing the well-founded semantics of [40].

In many cases, instead of checking if l is a consequence of *nlp* Π, we will be interested in finding answer sets of Π. This of course can be done only if Π has a finite Herbrand universe. There are various bottom up algorithms which can do such a computation rather efficiently for acyclic and stratified programs. As Theorem 7.3.3 shows, the answer set of an acyclic program can be also found by computing a classical model of propositional theory, $Comp(\Pi)$. The following generalization of the notion of acyclicity ensures one-to-one correspondence between the answer sets of an *nlp* Π and the models of its Clark's completion, and hence allows the use of efficient propositional solvers for computing answer sets of Π.

Definition 7.3.7 (Tight programs). *A nlp Π is called* tight *if there is a level mapping $\|\ \|$ of Π such that for every rule r of $gr(\Pi)$ and every $l \in pos(r)$, $\|head(r)\| > \|l\|$.*

It is easy to check that a program

$$\Pi_{17} \begin{cases} a \leftarrow b, \\ \qquad not\ a. \\ b. \end{cases}$$

is tight while program

$$a \leftarrow a$$

is not.

Theorem 7.3.4 (Second Fages' Theorem). *If nlp 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 [34]. In the last ten years a substantial amount of work was done to expand second Fages' theorem. One of the most important results in this direction is due to Fangzhen Lin and Yuting Zhao [58]. If program Π is tight then the corresponding propositional formula is simply the Clark's completion of Π; otherwise the corresponding formula is the conjunction of the completion of Π with the additional formulas that Lin and Zhao called the *loop formulas* of Π. The number of loop formulas is exponential in the size of Π in the worst case, and there are reasons for this in complexity theory [56]. But in many cases the Lin–Zhao translation of Π into propositional logic is not much bigger than Π. The reduction of the problem of computing answer sets to the satisfiability problem for propositional formulas given by the Lin–Zhao theorem has led to the development of answer set solvers such as ASET [58], CMODELS [3], etc. which are based on (possibly multiple) calls to propositional solvers.

Earlier solvers such as SMODELS and DLV compute answer sets of a program using substantially modified versions of Davis–Putnam algorithm, adopted for logic programs [55, 73, 72, 54]. All of these approaches are based on sophisticated methods for grounding logic programs. Even though the solvers are capable of working with hundreds of thousands and even millions of ground rules, the size of the grounding remains a major bottleneck of answer set solvers. There are new promising approaches to computing answer sets which combine Davis–Putnam like procedure with constraint satisfaction algorithms and resolution and only require partial grounding [31, 15]. We hope that this work will lead to substantial improvements in the efficiency of answer set solvers.

7.3.3 Properties of Entailment

Let us consider a program Π_{15} from Section 7.3. Its answer set is $\{a, c\}$ and hence both, a and c, are the consequences of Π_{15}. 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 the following condition

$$\frac{\Pi \models a, \quad \Pi \models b}{\Pi \cup \{a\} \models b} \tag{7.6}$$

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 Answer Set Prolog in which already proven lemmas are simply added to the program. There are, however, large classes of programs for which this problem does not exist.

Definition 7.3.8 (Cautious monotonicity). *We will say that a class of programs is* cautiously monotonic *if programs from this class satisfy condition* (7.6).

The following important theorem is due to H. Turner [92]

Theorem 7.3.5 (First Turner's Theorem). *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 (7.6) for order-consistent programs.

A much simpler observation guarantees that all *nlp*'s under the answer set semantics have the 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 *cumulativity*: augmenting a program with one of its consequences does not alter its consequences. More precisely,

Theorem 7.3.6 (Second Turner's Theorem). *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 cumulativity, cut, and cautious monotonicity were originally formulated for analysis of nonmonotonic consequence relations. Makinson's [59] handbook article includes a survey of such properties for nonmonotonic logics used in AI.

7.3.4 Relations between Programs

In this section we discuss several important relations between logic programs. We start with the notion of equivalence.

Definition 7.3.9 (Equivalence). *Logic programs are called equivalent if they have the same answer sets.*

It is easy to see, for instance, that programs

$$\Pi_{18} = \{p(a) \text{ } or \text{ } p(b)\},$$

$$\Pi_{19} = \{p(a) \leftarrow not\ p(b). \quad p(b) \leftarrow not\ p(a).\}$$

have the same answer sets, $\{p(a)\}$ and $\{p(b)\}$, and therefore are equivalent. Now consider programs Π_{20} and Π_{21} obtained by adding rules

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

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

to each of the programs Π_{18} and Π_{19}. It is easy to see that Π_{20} has one answer set, $\{p(a), p(b)\}$ while Π_{21} has no answer sets. The programs Π_{20} and Π_{21} are not equivalent. It is not of course surprising that, in general, epistemic disjunction cannot be eliminated from logic programs. As was mentioned before programs with and without *or* have different expressive powers. It can be shown, however, that for a large class of logic programs, called cycle-free [16], the disjunction can be eliminated by the generalization of the method applied above to Π_{18}. Program Π_{20} which does not belong to this class has a cycle (a mutual dependency) between elements $p(a)$ and $p(b)$ in the head of its rule. The above example suggests another important question: under what conditions can we be sure that replacing a part Π_1 of a knowledge base K by Π_2 will not change the answer sets of K? Obviously simple equivalence of Π_1 and Π_2 is not enough for this purpose. We need a stronger notion of equivalence [57].

Definition 7.3.10 (Strong equivalence). *Logic programs Π_1 and Π_1 with signature σ are called* strongly equivalent *if for every program Π with signature σ programs $\Pi \cup \Pi_1$ and $\Pi \cup \Pi_2$ have the same answer sets.*

The programs Π_{18} and

$$\Pi_{22} = \Pi_{18} \cup \{p(a) \leftarrow not\ p(b)\}$$

are strongly equivalent, while programs Π_{18} and Π_{19} are not. The notion of strong equivalence has deep and non-trivial connections with intuitionistic logics. One can show that if two programs in which *not*, *or*, and $\leftarrow$ are understood as intuitionistic negation, implication and disjunction, respectively, are intuitionistically equivalent, then they are also strongly equivalent. Furthermore in this statement intuitionistic logic can be replaced with a stronger subsystem of classical logic, called "the logic of here-and-there". Its role in logic programming was first recognized in [75], where it was used to define a nonmonotonic "equilibrium logic" which syntactically extends an original notion of a logic program. As shown in [57] two programs are equivalent iff they are equivalent in the logic of here-and-there.

There are other important forms of equivalence which were extensively studied in the last decade. Some of them weaken the notion of strong equivalence by limiting the class of equivalence preserving updates. For instance, programs Π_1 and Π_2 over signature σ are called *uniformly equivalent* if for any set of ground facts, F, of σ programs $\Pi_1 \cup F$ and $\Pi_2 \cup F$ have the same answer sets. Here the equivalence preserving updates are those which consist of collections of ground facts. It can be checked that programs Π_{18} and Π_{19}, while not strongly equivalent, are uniformly equivalent. Another way to weaken the original definition is to limit the signature of the updates. Programs Π_1 and Π_2 over signature σ are called *strongly equivalent relative to a given set A of ground atoms of σ* if for any program Π in the language of A, programs $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ have the same answer sets. Definition of the uniform equivalence can be relativized in a similar way. There is a substantial literature on the subject. As an illustration let us mention a few results established in [30]. We already mentioned that for head-cycle-free programs eliminating disjunction by shifting atoms from rule heads to the respective rule bodies preserves regular equivalence. In this paper the authors show that this transformation also preserves (relativized) uniform equivalence while it affects (relativized) strong equivalence. The systems for testing various forms of equivalence are described in [51].

7.4 A Simple Knowledge Base

To illustrate the basic methodology of representing knowledge in Answer Set Prolog, let us first consider a simple example from [43].

Example 7.4.1. 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 Answer Set 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:

$$\begin{array}{lll} member(sam, cs). & member(bob, cs). & member(tom, cs). \\ course(java, cs). & course(c, cs). & \\ course(ai, cs). & course(logic, cs). & \end{array} \tag{7.7}$$

together with the closed world assumptions expressed by the rules:

$$\begin{array}{ll} \neg member(P, cs) & \leftarrow not\ member(P, cs). \\ \neg course(C, cs) & \leftarrow not\ course(C, cs). \end{array} \tag{7.8}$$

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

$$teaches(sam, java). \quad teaches(bob, ai). \tag{7.9}$$

Since the schedule is incomplete, the relation *teaches* is open and the use of *CWA* for this relation 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*[2] which can be represented in Answer Set Prolog by the rules:

$$\begin{array}{ll} \neg teaches(P, C) & \leftarrow \neg member(P, cs), \\ & \quad course(C, cs), \\ & \quad not\ ab(d_1(P, C)), \\ & \quad not\ teaches(P, C). \\ ab(d_1(P, logic)) & \leftarrow not\ \neg member(P, math). \end{array} \tag{7.10}$$

[2]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

$$member(mary, math). \tag{7.11}$$

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 (7.10) can be replaced by a simpler rule

$$ab(d_1(P, logic)) \leftarrow member(P, math). \tag{7.12}$$

It is not difficult to show that the resulting programs have the same answer sets. To complete our definition of relation "*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:

$$\begin{aligned}\neg teaches(P_1, C) \leftarrow\; & teaches(P_2, C),\\ & P_1 \neq P_2,\\ & not\ ab(d_2(P_1, C)),\\ & not\ teaches(P_1, C).\end{aligned} \tag{7.13}$$

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

$$teaches(john, ai). \tag{7.14}$$

which defeats the corresponding conclusion of default (7.10). 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

$$\begin{aligned}& part(cs, cos).\\ & part(cos, u).\\ & part(E1, E2) \leftarrow part(E1, E),\\ & \qquad\qquad\qquad\quad part(E, E2).\\ & \neg part(E1, E2) \leftarrow not\ part(E1, E2).\end{aligned} \tag{7.15}$$

$$\begin{aligned}member(P, E1) \leftarrow\; & part(E2, E1),\\ & member(P, E2).\end{aligned} \tag{7.16}$$

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 (7.15) 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 rule:

$$\neg member(P, Y) \leftarrow not\ member(P, Y). \tag{7.17}$$

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 [72]).

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

$$member(sam, u)? \tag{7.18}$$

Asking DLV to answer $member(sam, u)?$ on program $\mathcal{K}$ we get precisely the response to our query under cautious entailment.[3] The answer set solvers also provide 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.

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

$$\begin{aligned} offered(C, D) \;&\leftarrow course(C, D), \\ & teaches(P, C). \\ \neg offered(C, D) &\leftarrow course(C, D), \\ & not\ offered(C, D). \end{aligned} \tag{7.19}$$

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

$$teaches(tom, logic)\ or\ teaches(bob, logic). \tag{7.20}$$

It is easy to see that the resulting program has two answer sets and that each answer set contains $offered(logic, cs)$. The example shows that Answer Set Prolog with disjunction allows a natural form of reasoning by cases—a mode of reasoning not easily modeled by Reiter's default logic. The answer sets of the new program can be computed by DLV and SMODELS based disjunctive answer set solver GnT [50]. It is worth noting that this program is head-cycle free and therefore, the disjunctive rule (7.20) can be replaced by two non-disjunctive rules,

$$\begin{aligned} teaches(tom, logic) &\leftarrow not\ teaches(bob, logic). \\ teaches(bob, logic) &\leftarrow not\ teaches(tom, logic). \end{aligned} \tag{7.21}$$

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.

7.5 Reasoning in Dynamic Domains

In this section we discuss the Answer Set Prolog representation of knowledge about *dynamic domains*. We assume that such a domain is modeled by a *transition diagram* with nodes corresponding to possible states of the domain, and arcs labeled by actions. An arc (σ_1, a, σ_2) indicates that execution of an action a in state σ_1 may result in the domain moving to the state σ_2. If for every state σ_1 and action a the diagram contains at most one arc (σ_1, a, σ_2) then the domain is called *deterministic*. The transition diagram contains all possible trajectories of the domain. Its particular history is given by a

[3] 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.

record of observations and actions. Due to the size of the diagram, the problem of finding its concise specification is not trivial and has been a subject of research for a comparatively long time. Its solution requires a good understanding of the nature of causal effects of actions in the presence of complex interrelations between fluents—propositions whose truth value may depend on the state of the domain. An additional level of complexity is added by the need to specify what is not changed by actions. The latter, known as the *frame problem* [48], is often reduced to the problem of finding a concise and accurate representation of the *inertia axiom*—a default which says that *things normally stay as they are*. The search for such a representation substantially influenced AI research during the last twenty years. An interesting account of history of this research together with some possible solutions can be found in [86].

To better understand the Answer Set Prolog way of specifying dynamic domains one may first look at a specification of such domains in the formalism of *action languages* (see, for instance, [46]). In this paper we limit our attention to an action description language AL from [12]. A *theory* of AL consists of a signature, Σ, and a collection of causal laws and executability conditions. The signature contains two disjoint, nonempty sets of symbols: the set F of fluents and the set A of *elementary actions*. A set $\{a_1, \dots, a_n\}$ of elementary actions is called a *compound* action. It is interpreted as a collection of elementary actions performed simultaneously. By *actions* we mean both elementary and compound actions. By *fluent literals* we mean fluents and their negations. By $\bar{l}$ we denote the fluent literal complementary to l. A set S of fluent literals is called *complete* if, for any $f \in F$, $f \in S$ or $\neg f \in S$. AL contains the following causal laws and executability conditions of the form

1. a_e *causes* l *if* p;
2. l *if* p;
3. *impossible* a *if* p

where a_e and a are elementary and arbitrary actions, respectively, and p is a collection of fluent literals from Σ, often referred to as the *precondition* of the corresponding law. If p is empty the *if* part of the propositions will be omitted. The first proposition, called *dynamic causal laws*, says that, if the elementary action a_e were to be executed in a state which satisfies p, the system will move to a state satisfying l. The second proposition, called a *static causal law*, says that every state satisfying p must satisfy l. The last proposition says that action a cannot happen in a state satisfying p. Notice that here a can be compound; *impossible* $(\{a_1, a_2\})$ means that elementary actions a_1 and a_2 cannot occur concurrently.

Let $\mathcal{A}$ be an action description of AL over signature Σ. To define the transition diagram, $T_{\mathcal{A}}$, described by $\mathcal{A}$ we need the following terminology and notation. Let S be a set of fluent literals of Σ. The set $Cn_{\mathcal{A}}(S)$ is the smallest set of fluent literals of Σ that contains S and satisfies static causal laws of $\mathcal{A}$. $E(a_e, \sigma)$ stands for the set of all fluent literals l for which there is a dynamic causal law "a_e *causes* l *if* p" in $\mathcal{A}$ such that $p \subseteq \sigma$. $E(a, \sigma) = \bigcup_{a_e \in a} E(a_e, \sigma)$. The transition system $T = \langle \mathcal{S}, \mathcal{R} \rangle$ *described* by an action description $\mathcal{A}$ is defined as follows:

1. $\mathcal{S}$ is the collection of all complete and consistent sets of fluent literals of Σ which satisfy static causal laws of $\mathcal{A}$;

2. $\mathcal{R}$ is the set of all triples (σ, a, σ') such that $\mathcal{A}$ does not contain a proposition of the form "*impossible a if p*" such that $p \subseteq \sigma$ and

$$\sigma' = Cn_{\mathcal{A}}(E(a, \sigma) \cup (\sigma \cap \sigma')). \tag{7.22}$$

   The argument of $Cn_{\mathcal{A}}$ in (7.22) is the union of the set $E(a, \sigma)$ of the "direct effects" of action a with the set $\sigma \cap \sigma'$ of facts that are "preserved by inertia". The application of $Cn_{\mathcal{A}}$ adds the "indirect effects" to this union.

The above definition is from [63] and is the product of a long investigation of the nature of causality. (An action language based on this definition appeared in [91].) Theorem 7.5.1 [6] (a version of the result from [91]) shows the remarkable relationship between causality expressible in *AL* and beliefs of rational agents as captured by the notion of answer sets of logic programs.

To formulate the theorem we will need some terminology. We start by describing an encoding τ of causal laws of *AL* into a program of Answer Set Prolog suitable for execution by answer set solvers:

1. $\tau(a_e \textit{ causes } l_0 \textit{ if } l_1 \ldots l_n)$ is the collection of atoms
   $dynamic\_law(d)$, $head(d, l_0)$, $action(d, a_e)$,
   $prec(d, i, l_i)$ for $1 \leqslant i \leqslant n$,
   $prec(d, n + 1, nil)$.
   Here d is a new term used to name the corresponding law, and *nil* is a special fluent constant. The last statement, $prec(d, n + 1, nil)$, is used to specify the end of the list of preconditions. (This arrangement simplifies the definition of relation $prec\_h(D, T)$ which holds when all the preconditions of default D hold at time step T.)

2. $\tau(l_0 \textit{ if } l_1 \ldots l_n)$ is the collection of atoms
   $static\_law(d)$, $head(d, l_0)$,
   $prec(d, i, l_i)$ for $1 \leqslant i \leqslant n$,
   $prec(d, n + 1, nil)$.

3. $\tau(\textit{impossible } \{a_1, \ldots, a_k\} \textit{ if } l_1 \ldots l_n)$ is a constraint

$$\leftarrow h(l_1, T), \ldots, h(l_n, T), occurs(a_1, T), \ldots, occurs(a_k, T).$$

Here T ranges over non-negative integers, $occurs(a, t)$ says that action a occurred at moment t, and $h(l, t)$ means that fluent literal l holds at t. (More precisely, $h(p(\bar{t}), T)$ stands for $holds(p(\bar{t}), T)$, while $h(\neg p(\bar{t}), T)$ is a shorthand for $\neg holds(p(\bar{t}), T)$. If σ is a collection of literals then $h(\sigma, T) = \{h(l, T)\colon\ l \in \sigma\}$. Finally, for any action description $\mathcal{A}$

$$\tau(\mathcal{A}) = \{\tau(law)\colon\ law \in \mathcal{A}\}, \tag{7.23}$$

$$\phi(\mathcal{A}) = \tau(\mathcal{A}) \cup \Pi(1), \tag{7.24}$$

$$\phi_n(\mathcal{A}) = \tau(\mathcal{A}) \cup \Pi(n), \tag{7.25}$$

where $\Pi(1)$ is an instance of the following program

$$\Pi(n)\begin{cases}
1.\ h(L,T') & \leftarrow \mathit{dynamic\_law}(D),\\
 & \quad \mathit{head}(D,L),\\
 & \quad \mathit{action}(D,A),\\
 & \quad \mathit{occurs}(A,T),\\
 & \quad \mathit{prec\_h}(D,T).\\
2.\ h(L,T) & \leftarrow \mathit{static\_law}(D),\\
 & \quad \mathit{head}(D,L),\\
 & \quad \mathit{prec\_h}(D,T).\\
3.\ \mathit{all\_h}(D,K,T) & \leftarrow \mathit{prec}(D,K,\mathit{nil}).\\
4.\ \mathit{all\_h}(D,K,T) & \leftarrow \mathit{prec}(D,K,P),\\
 & \quad h(P,T),\\
 & \quad \mathit{all\_h}(D,K',T).\\
5.\ \mathit{prec\_h}(D,T) & \leftarrow \mathit{all\_h}(D,1,T).\\
6.\ h(L,T') & \leftarrow h(L,T),\\
 & \quad \mathit{not}\ h(\overline{L},T').
\end{cases}$$

Here D, A, L are variables for the names of laws, actions, and fluent literals, respectively, T, T' are consecutive time points from interval $[0,n]$ and K, K' stand for consecutive integers used to enumerate preconditions of causal laws of $\mathcal{A}$. The first two rules describe the meaning of dynamic and static causal laws, rules (3), (4), (5) define what it means for all the preconditions of law D to succeed, and rule (6) represents the inertia axiom.

Theorem 7.5.1. *For any action description $\mathcal{A}$ of AL the transition diagram $T_{\mathcal{A}}$ contains a link (σ, a, σ') iff there is an answer set S of logic program*

$$\phi(\mathcal{A}) \cup h(\sigma,0) \cup \{\mathit{occurs}(a_i,0)\colon\ a_i \in a\}$$

such that, $\sigma' = \{l\colon\ h(l,1) \in S\}$.

The theorem establishes a close relationship between the notion of causality and the notion of rational beliefs of an agent.

This and similar results are used as a basis for the answer set planning, diagnostics, learning, etc. Consider for instance an action description $\mathcal{A}$ which contains a collection of elementary actions $e_0, \dots, e_n$ which can be performed by an intelligent agent associated with the domain. Let us assume that the transition system $T_{\mathcal{A}}$ is deterministic, i.e., any state σ and action a have at most one successor state. The agent, who is currently in a state σ, needs to find a sequential plan of length k to achieve a goal $g = \{l_1, \dots, l_m\}$. In other words the agent is looking for a trajectory $\langle \sigma, e_0, \dots, e_{k_1}, \sigma' \rangle$ of $T_{\mathcal{A}}$ where $g \subseteq \sigma'$. Using Theorem 7.5.1 it is not difficult to show that there is one to one correspondence between such trajectories and answer sets of the program

$$\mathit{pl}(\mathcal{A},k) = \phi(n) \cup h(\sigma,0) \cup P_M,$$

where

$$P_M \begin{cases} occurs(e, T) \text{ or } \neg occurs(e, T) \leftarrow T < k. \\ \neg occurs(e_2, T) \leftarrow occurs(e_1, T), e_1 \neq e_2. \\ goal \leftarrow h(g, k). \\ \leftarrow not\ goal. \end{cases}$$

The first two rules guarantee the occurrence of exactly one agent's action at each time step of the trajectory. The next two ensure that every answer set of the program satisfies the goal at step k. The correspondence allows to reduce the problem of classical planning to the problem of finding answer sets of logic programs. A simple loop calls an answer set solver with the program $pl(\mathcal{A}, i)$ as an input for i ranging from 0 to k. A plan is easily extracted from the first answer set returned by the solver. If no answer set is found then the planning problem has no solution of the length less than or equal to k. The method, first suggested in [90, 26], has a number of practical applications [74] and in some cases may be preferable to other approaches. Typical classical planners for instance do not allow the input language with static causal laws, which can be essential for modeling some domains, as well as for efficiency of planning. Moreover such planners may require special languages describing properties of the plans, etc. To illustrate this point let us consider a complex hydraulic module from the reaction control system of the space shuttle. In a very simplified view the system can be viewed as a graph whose nodes are labeled by tanks containing propellant, jets, junctions of pipes, etc. Arcs of the graph are labeled by valves which can be open or closed by a collection of switches. The goal is to open or close valves to deliver propellant from tanks to a proper combination of jets. The graph can be described by a collection of atoms of the form $connected(n_1, v, n_2)$—valve v labels the ark from n_1 to n_2—and $controls(s, v)$—switch s controls the valve v. The description of the system may also contain a collection of faults, e.g., $stuck(V)$, which indicates that valve V is stuck. We assume that our information about malfunctioning of valves is complete, i.e.,

$$\neg stuck(V) \leftarrow not\ stuck(V).$$

The domain contains actions $flip(S)$. The dynamic causal laws for this action are given by the rules

$$\begin{aligned} h(state(S, open), T+1) &\leftarrow occurs(flip(S), T), \\ &\quad\ h(state(S, closed), T). \\ h(state(S, closed), T+1) &\leftarrow occurs(flip(S), T), \\ &\quad\ h(state(S, open), T). \end{aligned}$$

The next rule is a static causal law describing the connections between positions of switches and valves.

$$\begin{aligned} h(state(V, P), T) \leftarrow\ & controls(S, V), \\ & h(state(S, P), T), \\ & \neg stuck(V). \end{aligned}$$

The next static causal law describes the relationship between the values of fluent $pressurized(N)$ for neighboring nodes.

$$\begin{aligned} h(pressurized(N_2), T) \leftarrow\ & connected(N_1, V, N_2), \\ & h(pressurized(N_1), T), \\ & h(state(V, open), T). \end{aligned}$$

We also assume that tanks are always pressurized which will be encoded by the rule

$$h(pressurized(N), T) \leftarrow tank(N).$$

The laws describe a comparatively complex effect of a simple *flip* operation which propagates the pressure through the system. It seems that without static causal laws the substantially longer description will be needed to achieve this goal. Suppose now that some of the valves may be leaking. It is natural to look for plans which do not open leaking valves. This can be achieved by expanding the standard planning module by the rule

$$\begin{aligned}\neg occurs(flip(S), T) \leftarrow\; & controls(S, V),\\ & h(state(S, closed), T),\\ & is\_leaking(V).\end{aligned}$$

Adding the rule

$$\begin{aligned}\neg occurs(flip(S), T) \leftarrow\; & controls(S, V),\\ & stuck(V).\end{aligned}$$

will help to avoid generation of unnecessary actions, etc. These and similar rules can be used to improve quality of plans and efficiency of the planner. It is also worth noticing that simple modification of the planner will allow search for parallel plans, that similar techniques can be used to search for conformant and conditional plans [10, 89], for diagnostics [6] and even for learning [5, 81].

7.6 Extensions of Answer Set Prolog

In this section we briefly discuss extensions of Answer Set Prolog by aggregates [32] and by consistency restoring rules [7]. To see the need for the first extension let us consider the following example.

Example 7.6.1. Suppose that we are given a complete collection of records

$$\begin{array}{ll} registered(john, cs1). & registered(mary, cs2).\\ registered(bob, cs1). & registered(sam, cs2).\\ registered(mike, cs1). & \end{array}$$

and that our goal is to define the notion of a large class—a class with at least three registered students. In the language DLP^A from [32] this can be done by the rule

$$large\_class(C) \leftarrow \#count(\{S : registered(S, C)\}) \geqslant 3.$$

Here $\#count(X)$ is the cardinality of the set X. Clearly $\#count(\{S\colon registered(S, cs1)\}) = 3$ and hence $cs1$ is a large class.

The syntax of DLP^A [4] allows *aggregate atoms* of the form $f(\{X : p(X)\})\ \mathrm{rel}\ n$ where *rel* is a standard arithmetic relation and n is a number. The occurrence of variable X in the above aggregate atom is called *bound*. Such occurrences remain

[4] For simplicity we omit several less important features of the language.

untouched by the grounding process. Rules of DLP^A are of the form

$$a_1 \; or \; \dots \; or \; a_n \leftarrow b_1, \dots, b_k, not \; b_{k+1}, \dots, not \; b_m,$$

where a's are standard (non-aggregate) atoms and b's are atoms. The program, P_0, from Example 7.6.1 is a ground program of DLP^A.

Let S be a set of standard ground atoms from the signature of a DLP^A program P.

Definition 7.6.1 (Answer sets of DLP^A). *An aggregate atom $f(\{X : p(X)\})$ rel n is* true *in S if $f(\{X : p(X) \in S\})$ rel n; it is false otherwise. The* DLV^A reduct, $P^{[S]}$ *of P with respect to S is obtained from $gr(P)$ by removing all rules whose bodies contain extended literals which are false in S. S is an* answer set *of P if it is a minimal set closed under the rules of $P^{[S]}$.*

For programs not containing aggregate atoms the definition is equivalent to the original definition of answer sets. It is easy to check that program P_0 from Example 7.6.1 has unique answer set consisting of the facts of the program and the atom *large_class*$(cs1)$. The next two programs illustrate the DLV^A treatment of recursion through aggregates. Such a recursion caused various difficulties for a number of other approaches to expending logic programs with aggregates. Let

$$\Pi_1 = \{p(a) \leftarrow \#count(\{X : p(X)\}) > 0.\}$$

and

$$\Pi_2 = \{p(a) \leftarrow \#count(\{X : p(X)\}) < 1.\}$$

and consider sets $S_1 = \{p(a)\}$ and $S_2 = \emptyset$. Since $\Pi_1^{[S_1]} = \Pi_1$ and $\emptyset$ is closed under Π_1, S_1 is not an answer set of Π_1. But $\Pi_1^{[S_2]} = \emptyset$ and hence S_2 is the only answer set of Π_1. Since $\Pi_2^{[S_1]} = \emptyset$ and $\Pi_2^{[S_2]} = \Pi_2$ program Π_2 has no answer sets.

Now we give a brief description of CR-Prolog—an extension of Answer Set Prolog capable of encoding rare events. We start with a description of syntax and semantics of the language. For simplicity we omit the CR-Prolog treatment of preferences.

A program of CR-Prolog is a pair consisting of signature and a collection of regular rules of Answer Set Prolog and rules of the form

$$l_0 \stackrel{+}{\leftarrow} l_1, \dots, l_k, not \; l_{k+1}, \dots, not \; l_n \tag{7.26}$$

where l's are literals. Rules of type (7.26) are called *consistency restoring* rules (cr-rules). Intuitively the rule says that if the reasoner associated with the program believes the body of the rule then it "may possibly" believe one element of the head. This possibility however may be used only if there is no way to obtain a consistent set of beliefs by using only regular rules of the program.

The set of regular rules of a CR-Prolog-program Π will be denoted by Π^r; the set of cr-rules of Π will be denoted by Π^{cr}. By $\alpha(r)$ we denote a regular rule obtained from a consistency restoring rule r by replacing $\stackrel{+}{\leftarrow}$ by $\leftarrow$; α is expended in a standard way to a set R of cr-rules. As usual, the semantics of CR-Prolog will be given for ground programs, and a rule with variables will be viewed as a shorthand for schema of ground rules.

Definition 7.6.2 (Answer sets of CR-Prolog). *A minimal* (*with respect to set theoretic inclusion*) *collection R of cr-rules of Π such that $\Pi^r \cup \alpha(R)$ is consistent* (*i.e. has an answer set*) *is called an* abductive support *of Π.*

A set A is called an answer set *of Π if it is an answer set of a regular program $\Pi^r \cup \alpha(R)$ for some abductive support R of Π.*

Example 7.6.2. Consider a program, T, of CR-Prolog consisting of rules

$$\begin{array}{lll} p(X) & \leftarrow & not\ ab(X). \\ ab(e1). & & \\ ab(e2). & & \\ q(e). & & \\ r(X) & \leftarrow & p(X), q(X). \\ ab(X) & +\!\!- . & \end{array}$$

The program includes a default with two exceptions, a partial definition of r in terms of p and q, and consistency restoring rule which acknowledges the possibility of existence of unknown exceptions to the default. Since normally such a possibility is ignored the answer set of the program consists of its facts and atoms $p(e), r(e)$.

Suppose now that the program is expanded by a new atom, $\neg r(e)$. The regular part of the new program has no answer set. The cr-rule solves the problem by assuming that e is a previously unknown exception to the default. The resulting answer set consists of the program facts and the atom $ab(e)$.

The possibility to encode rare events which may serve as unknown exceptions to defaults proved to be very useful for various knowledge representation tasks, including planning, diagnostics, and reasoning about the agents intentions [4, 13].

7.7 Conclusion

We hope that the material is this chapter is sufficient to introduce the reader to Answer Set Prolog, its mathematical theory, and its applications. We will conclude by briefly outlining the relationship between this formalism and other areas of Knowledge Representation presented in this book. The semantics of the language has its roots in nonmonotonic logics discussed in Chapter 6. The original intuition of stable model semantics comes from the mapping of logic programming rules into formulas of Moore's autoepistemic logic [68]. The mapping, first presented in [41], interprets default negation, *not* p, of Prolog as $\neg Lp$ where L is the belief operator of Autoepistemic Logic. This interpretation is responsible for the epistemic character of the stable model semantics. In [17, 60, 45] logic programs with classical negation (but without disjunction) were mapped into Reiter's Default Theory [79]. Very close relationship between Answer Set Prolog and Circumscription [64] was recently established in [36]. There is also a close relationship between Answer Set Prolog and Causal Logic discussed in Chapter 19. As was discussed in Section 7.3.2 computational methods of ASP are closely related to topics discussed in chapters on satisfiability and constraint programming. The designers of ASP solvers commonly use ideas from these areas. The additional power of ASP, its ability to represent transitive closure, aggregates, and other features not immediately available in satisfiability solvers, together with sophisticated grounding methods can undoubtedly be useful for the SAT community. Planning and diagnostic algorithms based on ASP can nicely complement more traditional planning methods discussed in Chapter 22. These methods are especially useful when successful planning requires a large body of knowledge and when the agent needs to solve both, planning and diagnostic, problems. It is our hope that the ongoing work on combining the traditional ASP methods with constraint programming algorithms will help to overcome the limitations caused by grounding, and lead to the development of efficient planning and scheduling systems. The methodology of modeling dynamic systems in Answer Set Prolog discussed in Section 7.5 has much in common with other model-based problem solving methods of Chapter 10. It will be interesting to investigate the range of applicability and advantages and disadvantages of various styles of description of states and possible trajectories of the domain, and of reasoning methods used in model-based problem solving. There is also a substantial cross-fertilization between answer set based reasoning about actions and change and other similar formalisms including Situation Calculus [48, 80], Event Calculus [53, 69], and various temporal logics. There are, for instance, logic programming based counterparts of Situation Calculus, which allow elegant solutions to the frame and ramification problem. Original versions of Event Calculus were directly expressed in the language of logic programming. The ability of temporal logic to reason about properties of paths is modeled by logic programming based specification of goals in [8]. Chapter 20 gives an example of the use of Answer Set Prolog and its reasoning methods for representing and reasoning about commonsense and linguistic knowledge needed for intelligent question answering from natural language texts. There are several interesting efforts of combining Answer Sets with Bayesian net based probabilistic reasoning, which substantially increases expressive power of both knowledge representation languages and promises to lead to efficient algorithms for answering some forms of probabilistic queries. Finally, new results establishing some relationship between Description Logic and Answer Sets (see, for instance, [28]) may open the way for interesting applications of Answer Sets to Semantic Web.

Acknowledgements

This work was partially supported by ARDA grant ASU06-C-0143 and NASA grant NASA-NNG05GP48G. The author wish to thank Gerhard Brewka, Vladimir Lifschitz, and Hudson Turner for many useful comments.

Bibliography

[1] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[2] K. Apt and D. Pedreschi. Proving termination in general Prolog programs. In *Proc. of the Internat. Conf. on Theoretical Aspects of Computer Software*, *LNCS*, vol. 526, pages 265–289. Springer-Verlag, 1991.

[3] Y. Babovich and M. Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR-05*, Jan. 2004.

[4] M. Balduccini. USA-Smart: Improving the quality of plans in answer set planning. In *PADL'04*, *Lecture Notes in Artificial Intelligence (LNCS)*. Springer, June 2004.

[5] M. Balduccini. Answer set based design of highly autonomous, rational agents. PhD thesis, Texas Tech University, Dec. 2005.

[6] M. Balduccini and M. Gelfond. Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP)*, 3(4–5):425–461, July 2003.

[7] M. Balduccini and M. Gelfond. Logic programs with consistency-restoring rules. In P. Doherty, J. McCarthy, and M.-A. Williams, editors, *International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2003 Spring Symposium Series*, pages 9–18, March 2003.

[8] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving with Answer Sets*. Cambridge University Press, 2003.

[9] C. Baral, K. Chancellor, N. Tran, A. Joy, and M. Berens. A knowledge based approach for representing and reasoning about cell signalling networks. In *Proceedings of European Conference on Computational Biology, Supplement on Bioinformatics*, pages 15–22, 2004.

[10] C. Baral, T. Eiter, and Y. Zhao. Using SAT and logic programming to design polynomial-time algorithms for planning in non-deterministic domains. In *Proceedings of AAAI-05*, pages 575–583, 2005.

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

[12] C. Baral and M. Gelfond. Reasoning agents in dynamic domains. In *Workshop on Logic-Based Artificial Intelligence*. Kluwer Academic Publishers, June 2000.

[13] C. Baral and M. Gelfond. Reasoning about intended actions. In *Proceedings of AAAI05*, pages 689–694, 2005.

[14] C. Baral, M. Gelfond, and N. Rushton. Probabilistic reasoning with answer sets. In *Proceedings of LPNMR-7*, January 2004.

[15] S. Baselice, P.A. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In *Proceedings of ICLP-05*, pages 52–66, 2005.

[16] R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

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

[18] G. Brewka. Logic programming with ordered disjunction. In *Proceedings of AAAI-02*, pages 100–105, 2002.

[19] G. Brewka. Answer sets: From constraint programming towards qualitative optimization. In *Proc. of 7th International Conference on Logic Programming and Non Monotonic Reasoning (LPNMR-04)*, pages 34–46. Springer-Verlag, Berlin, 2004.

[20] D.R. Brooks, E. Erdem, J.W. Minett, and D. Ringe. Character-based cladistics and answer set programming. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages*, pages 37–51, 2005.

[21] F. Buccafurri, N. Leone, and P. Rullo. Adding weak constraints to disjunctive datalog. In *Proceedings of the 1997 Joint Conference on Declarative Programming APPIA-GULP-PRODE'97*, 1997.

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

[23] K. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

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

[25] J.P. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20(2):308–334, 2004.

[26] Y. Dimopoulos, J. Koehler, and B. Nebel. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the 4th European Conference on Planning*, *Lecture Notes in Artificial Intelligence (LNCS)*, vol. 1348, pages 169–181. Springer, 1997.

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

[28] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. Technical Report INFSYS RR-1843-07-04, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria, January 2007. Preliminary version appeared in *Proc. KR 2004*, pages 141–151.

[29] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A deductive system for nonmonotonic reasoning. In *International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR97*, *LNAI*, vol. 1265, pages 363–374. Springer-Verlag, Berlin, 1997.

[30] T. Eiter, S. Woltran, and M. Fink. Semantical characterizations and complexity of equivalences in answer set programming. *ACM Transactions on Computational Logic*, 2006.

[31] I. Elkabani, E. Pontelli, and T.C. Son. Smodels with CLP and its applications: A simple and effective approach to aggregates in asp. In *Proceedings of ICLP-04*, pages 73–89, 2004.

[32] W. Faber. Unfounded sets for disjunctive logic programs with arbitrary aggregates. In *In Proc. of 8th International Conference on Logic Programming and Non Monotonic Reasoning (LPNMR 2005)*, *LNAI*, vol. 3662, pages 40–52. Springer-Verlag, Berlin, 2005.

[33] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2004)*, pages 200–212, 2004.

[34] 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.

[35] P. Ferraris. Answer sets for propositional theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LP-NMR)*, pages 119–131, 2005.

[36] P. Ferraris, J. Lee, and V. Lifschitz. A new perspective on stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 372–379, 2007.

[37] P. Ferraris and V. Lifschitz. Mathematical foundations of answer set programming. In *We Will Show Them, Essays in Honour of Dov Gabbay*, vol. 1, pages 615–654. College Publications.

[38] P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

[39] M. Gebser, B. Kaufman, A. Neumann, and T. Schaub. Conflict-deriven answer set enumeration. In C. Baral, G. Brewka, and J. Schlipf, editors. *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, *LNAI*, vol. 3662, pages 136–148. Springer, 2007.

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

[41] M. Gelfond. On stratified autoepistemic theories. In *Proceedings of Sixth National Conference on Artificial Intelligence*, pages 207–212, 1987.

[42] M. Gelfond. Epistemic approach to formalization of commonsense reasoning. Technical Report TR-91-2, University of Texas at El Paso, 1991.

[43] M. Gelfond and N. Leone. Knowledge representation and logic programming. *Artificial Intelligence*, 138(1–2), 2002.

[44] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.

[45] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3–4):365–386, 1991.

[46] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16):193–210, 1998.

[47] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36:345–377, 2006.

[48] P.J. Hayes and J. McCarthy. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors. *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[49] S. Heymans, D.V. Nieuwenborgh, and D. Vermeir. Guarded open answer set programming. In *Proc. of 8th International Conference on Logic Programming and Non Monotonic Reasoning (LPNMR 2005)*, *LNAI*, vol. 3662, pages 92–104. Springer-Verlag, Berlin, 2005.

[50] T. Janhunen, I. Niemela, P. Simons, and J. You. Partiality and disjunction in stable model semantics. In *Proceedings of the 2000 KR Conference*, pages 411–419, 2000.

[51] T. Janhunen and E. Oikarinen. LPEQ and DLPEQ—translators for automated equivalence testing of logic programs. In *Proc. of 8th International Conference on Logic Programming and Non Monotonic Reasoning (LPNMR 2004)*, *LNAI*, vol. 2923, pages 336–340. Springer-Verlag, Berlin, 2004.

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

[53] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(4):319–340, 1986.

[54] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2006.

[55] N. Leone, P. Rullo, and F. Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation*, 135:69–112, 1997.

[56] V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7(2):261–268, 2006.

[57] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.

[58] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.

[59] D. Makinson. General patterns in nonmonotonic reasoning. In *The Handbook on Logic in AI and Logic Programming*, vol. 3, pages 35–110. Oxford University Press, 1993.

[60] V.W. Marek and M. Truszczynski. Stable semantics for logic programs and default reasoning. In *Proc. of the North American Conf. on Logic Programming*, pages 243–257, 1989.

[61] V.W. Marek and M. Truszczynski. *Nonmonotonic Logics; Context Dependent Reasoning*. Springer-Verlag, Berlin, 1993.

[62] V.W. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, Berlin, 1999.

[63] N. McCain and H. Turner. A causal theory of ramifications and qualifications. In *Proceedings of IJCAI-95*, pages 1978–1984, 1995.

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

[65] J. McCarthy. In V. Lifschitz, editor. *Formalization of Common Sense*. Ablex, 1990.

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

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

[68] R.C. Moore. Semantical considerations on nonmonotonic logic. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 272–279. Morgan Kaufmann, August 1983.

[69] E.T. Mueller. *Commonsense Reasoning*. Morgan Kaufmann, 2006.

[70] I. Niemela. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–247, 1999.

[71] I. Niemela and P. Simons. Smodels—an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, *Lecture Notes in Artificial Intelligence (LNCS)*, vol. 1265, pages 420–429. Springer, 1997.

[72] I. Niemela, P. Simons, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, June 2002.

[73] I. Niemela and P. Simons. Smodels—an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the*

4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97), *Lecture Notes in Artificial Intelligence (LNCS)*, vol. 1265, pages 420–429. Springer, 1997.

[74] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog decision support system for the Space Shuttle. In *PADL 2001*, pages 169–183, 2001.

[75] D. Pearce. A new logical characterization of stable models and answer sets. In *Non-Monotonic Extension of Logic Programming*, *Lecture Notes in Artificial Intelligence (LNCS)*, vol. 1216, pages 57–70. Springer-Verlag, 1997.

[76] N. Pelov, M. Denecker, and M. Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*, 7:355–375, 2007.

[77] 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, 1988.

[78] R. Reiter. On closed world data bases. In *Logic and Data Bases*, pages 119–140. Plenum Press, 1978.

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

[80] R. Reiter. *Knowledge in Action—Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, September 2001.

[81] C. Sakama. Induction from answer sets in nonmonotonic logic programs. *ACM Transactions on Computational Logic*, 6(2):203–231, April 2005.

[82] C. Sakama and K. Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence*, 123:185–222, 2000.

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

[84] J. Schlipf. Some remarks on computability and open domain semantics. In *Proceedings of the Workshop on Structural Complexity and Recursion-Theoretic Methods in Logic Programming of the International Logic Programming Symposium*, 1993.

[85] J. Schlipf. The expressive powers of logic programming semantics. *Journal of Computer and System Sciences*, 51(1):64–86, 1995.

[86] M. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Commonsense Law of Inertia*. MIT Press, 1997.

[87] T. Soininen and I. Niemela. Developing a declarative rule language for applications in product configuration. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages*, pages 305–319, 1998.

[88] T.C. Son and E. Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, 7:355–375, 2007.

[89] T.C. Son, P.H. Tu, M. Gelfond, and A.R. Morales. An approximation of action theories of and its application to conformant planning. In *Proc. of 8th International Conference on Logic Programming and Non Monotonic Reasoning (LPNMR 2005)*, *LNAI*, vol. 3662, pages 172–184. Springer-Verlag, Berlin, 2005.

[90] V.S. Subrahmanian and C. Zaniolo. Relating stable models and AI planning domains. In *Proceedings of ICLP-95*, pages 233–247, 1995.

[91] H. Turner. Representing actions in logic programs and default theories: A situation calculus approach. *Journal of Logic Programming*, 31(1–3):245–298, June 1997.

[92] H. Turner. Order-consistent programs are cautiously monotonic. *Journal of Theory and Practice of Logic Programming (TPLP)*, 1(4):487–495, 2001.

[93] J. Vaghani, K. Ramamohanarao, D.B. Kemp, Z. Somogyi, P.J. Stuckey, T.S. Leask, and J. Harland. The Aditi deductive database system. *The VLDB Journal*, 3(2):245–288, 1994.