Introduction to Answer Set Programming

Vladimir Lifschitz
University of Texas at Austin

DRAFT

Answer set programming (ASP) is a form of declarative programming oriented towards difficult combinatorial search problems. It has been applied, for instance, to plan generation and product configuration problems in artificial intelligence and to graph-theoretic problems arising in VLSI design and in historical linguistics.

Syntactically, ASP programs look like Prolog programs, but the computational mechanisms used in ASP are different: they are based on the ideas that have led to the development of fast satisfiability solvers for propositional logic.

Answer set programming has emerged from interaction between two lines of research—on the semantics of negation in logic programming [Gelfond and Lifschitz, 1988] and on applications of satisfiability solvers to search problems [Kautz and Selman, 1992]. It was identified as a new programming paradigm in [Lifschitz, 1999, Marek and Truszczyński, 1999, Niemelä, 1999].

The central part of this survey is Section 5, where we discuss examples and the methodology of answer set programming. The other parts of the survey provide the necessary background (Sections 1–4) and describe some of the mathematical principles behind the design of answer set solvers (Sections 6, 7).

1 Logic Programs and Answer Sets

The syntax of formulas, rules and logic programs is defined as follows. *Formulas* are formed from propositional atoms and 0-place connectives $\top$ and $\bot$ using negation (written as *not*), conjunction (written as a comma) and disjunction (written as a semicolon). In multiple conjunctions

$$(\cdots((F_1, F_2), F_3), \ldots, F_{m-1}), F_m$$

and disjunctions

$$(\cdots((F_1; F_2); F_3); \ldots; F_{m-1}); F_m$$

parentheses can be dropped. If $m = 0$ then $F_1, \ldots, F_m$ is understood as $\top$ and $F_1; \ldots; F_m$ is understood as $\bot$.

A *rule* is an expression of the form

$$F \leftarrow G$$

(read "F if G"), where F and G are formulas, called the *head* and the *body* of the rule. A rule of the form $F \leftarrow \top$ will be identified with the formula F.

A *(logic) program* is a finite set of rules.[1]

The definition of an answer set for an arbitrary program will be given in the next section. Here we start by defining this concept for programs whose rules have the form

$$A_0 \leftarrow A_1, \ldots, A_m \tag{1}$$

where $m \geq 0$ and $A_0, \ldots, A_m$ are atoms. First, an auxiliary definition: a set X is said to *satisfy* such a program Π if, for every rule (1) in Π, $A_0 \in X$ whenever $A_1, \ldots, A_m \in X$. For instance, any program of this kind is satisfied by the set of the heads A_0 of all its rules (1).

For any program Π whose rules have form (1), the intersection of all sets that satisfy Π satisfies Π also. This fact allows us to talk about the *smallest* set of atoms that satisfies Π [van Emden and Kowalski, 1976]. This set is called *the answer set* for Π.

Consider, for instance, the program

$$\begin{array}{l} p \\ r \leftarrow p, q. \end{array}$$

The sets of atoms satisfying this program are

$$\{p\},\ \{p, r\},\ \{p, q, r\},$$

and its answer set is $\{p\}$.

Now the definition of an answer set will be extended to programs consisting of rules of the form

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \tag{2}$$

[1] The definition of the syntax of logic programs adopted in this survey is similar to the definition from [Lifschitz *et al.*, 1999], but less general. The latter allows a program to consist of infinitely many rules (actually nothing in Sections 1–3 depends on the assumption that programs are finite). More importantly, it allows formulas to contain two kinds of negation, along the lines of [Gelfond and Lifschitz, 1991], which is useful in some applications of ASP [Lifschitz, 2002].

where $n \geq m \geq 0$ and $A_0, \ldots, A_n$ are atoms. The *reduct* Π^X of such a program Π relative to a set X of atoms is the set of rules

$$A_0 \leftarrow A_1, \ldots, A_m$$

for all rules (2) in Π such that $A_{m+1}, \ldots, A_n \notin X$. We say that X is an *answer set* for Π if X is the answer set for Π^X (that is, the smallest set of atoms satisfying Π^X).

This definition was proposed in [Gelfond and Lifschitz, 1988], where the term "stable model" is used instead of "answer set"; the idea comes from [Reiter, 1980]. The second definition of an answer set is a generalization of the first: if Π does not contain negation ($m = n$ in every rule (2) of the program) then the only answer set for Π is the smallest set of atoms satisfying Π.

Example: if Π is

$$\begin{array}{l} p \leftarrow \mathit{not}\ q \\ q \leftarrow \mathit{not}\ r \end{array} \tag{3}$$

and $X = \{q\}$ then the reduct Π^X consists of one rule q. Consequently, X is an answer set for Π.

Exercise 1.1 Check that $\{p\}$ is not an answer set for program (3).

Does program (3) have answer sets other than $\{q\}$? We can find this out by checking each of the remaining 6 subsets of $\{p, q, r\}$. General properties of answer sets show, however, that the result will be negative for each of them. First, an atom can belong to an answer set for a program Π only if it is the head A_0 of one of its rules (2). In case of program (3), this fact tells us that its answer sets do not contain r, so that we only need to check $\emptyset$ and $\{p, q\}$. Second, any program Π consisting of rules of form (2) has the "anti-chain property": one answer set for Π cannot be a proper part of another answer set for Π. Consequently, $\emptyset$ (which is a subset of $\{q\}$) and $\{p, q\}$ (which is a superset of $\{q\}$) cannot be answer sets for (3) either.

Exercise 1.2 Find an answer set for the program Π_n consisting of n rules

$$p_i \leftarrow \mathit{not}\ p_{i+1} \qquad (1 \leq i \leq n)$$

where n is a positive integer. (Program (3) is essentially Π_2.)

Each of the programs Π_n has actually a unique answer set. On the other hand, the program

$$\begin{array}{l} p \leftarrow \mathit{not}\ q \\ q \leftarrow \mathit{not}\ p \end{array} \tag{4}$$

has two answer sets: $\{p\}$ and $\{q\}$. The one-rule rogram

$$p \leftarrow not\ p \tag{5}$$

has no answer sets.

Exercise 1.3 Find all answer sets for the following program, which extends (4) by two additional rules:

$$\begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \\ r \leftarrow p \\ r \leftarrow q. \end{array}$$

Exercise 1.4 Find all answer sets for the following combination of programs (4) and (5):

$$\begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \\ r \leftarrow not\ r \\ r \leftarrow p. \end{array}$$

The definition of an answer set for rules of form (2) was originally proposed as a semantics of negation in Prolog, that is to say, as a possible definition of a "correct" answer to a query for Prolog programs with negation. Let Π be the set of rules of form (2) obtained from a given Prolog program by grounding (replacing each rule with variables by all its ground instances). If Π has a unique answer set X then the correct answer to a ground query A is *yes* or *no* depending on whether A belongs to X or not. For instance, given program (3), a Prolog system is supposed to answer *yes* to query q and *no* to queries p and r.

From this perspective, a program with several answer sets, such as (4), is "bad"—it does not provide an unambiguous specification for the behavior of a Prolog system. Programs without answer sets, such as (5), are "bad" also.

In answer set programming, on the other hand, programs without a unique answer set are quite useful. The idea of ASP is to represent the search problem we are interested in as the problem of finding an answer set for some logic program, and then find a solution using an answer set solver—a system for generating answer sets. ASP programs with several answer sets correspond to search problems that have several solutions. ASP programs without answer sets correspond to problems that have no solutions.

As a semantics of negation in Prolog, the concept of an answer set is just one of several possibilities discussed in the literature. Some of the others are program completion [Clark, 1978], iterated fixpoints [Apt *et al.*, 1988] and well-founded models [Van Gelder *et al.*, 1991].

2 General Definition of an Answer Set

To define the concept of an answer set for arbitrary programs, we first need to generalize the definitions of satisfaction and reduct given in Section 1.

The *satisfaction relation* $X \models F$ between a set X of atoms and a formula F is defined recursively, as follows:

- for an atom A, $X \models A$ if $A \in X$
- $X \models \top$
- $X \not\models \bot$
- $X \models (F, G)$ if $X \models F$ and $X \models G$
- $X \models (F; G)$ if $X \models F$ or $X \models G$
- $X \models not\ F$ if $X \not\models F$.

We say that X *satisfies* a program Π (symbolically, $X \models \Pi$) if, for every rule $F \leftarrow G$ in Π, $X \models F$ whenever $X \models G$.

It is easy to see that this is a generalization of the definition of satisfaction from Section 1: if the rules of Π have the form (1) then X satisfies Π in the sense of this definition iff X satisfies Π in the sense of Section 1. On the other hand, our definition of $\models$ is a special case of the usual definition of satisfaction for propositional formulas, in disguise. Indeed, every rule $F \leftarrow G$ can be viewed as a propositional formula: rewrite it as the implication $G \supset F$ and replace each *not* in F and G with $\neg$, each comma with $\wedge$, and each semicolon with $\vee$. Every set X of atoms can be viewed as a truth assignment—as the function that maps the elements of X to t and all other atoms to f. Then a set X of atoms satisfies a program Π iff the truth assignment corresponding to X satisfies, in the sense of propositional logic, the set of formulas corresponding to the rules of Π. For instance, set $\{p\}$ satisfies program (4); accordingly, the truth assignment that maps p to t and q to f satisfies the propositional formulas

$$\begin{array}{l}\neg q \supset p\\ \neg p \supset q.\end{array}$$

The *reduct* F^X of a formula F with respect to a set X of atoms is defined recursively, as follows:

- if F is an atom or a 0-place connective then $F^X = F$
- $(F, G)^X = F^X, G^X$
- $(F; G)^X = F^X; G^X$
- $(not\ F)^X = \begin{cases} \bot\,, & \text{if } X \models F, \\ \top\,, & \text{otherwise.} \end{cases}$

The *reduct* Π^X of a program Π with respect to X is the set of rules

$$F^X \leftarrow G^X$$

for all rules $F \leftarrow G$ in Π.

In application to programs consisting of rules of form (2), the reduct in the sense of this definition is, strictly speaking, not quite the same as the reduct in the sense of Section 1. For instance, according to the new definition, the reduct of (3) relative to $\{q\}$ is the program

$$\begin{array}{l} p \leftarrow \bot \\ q \leftarrow \top \end{array} \tag{6}$$

which is not the same as the reduct q that we found in Section 1. But this is not an essential difference, because q and (6) are satisfied by the same sets of atoms. This is an instance of a general fact: for any program consisting of rules of form (2) and any set X of atoms, the reduct Π^X as defined here is a program satisfied by exactly the same sets of atoms as the reduct in the sense of Section 1.

Now we are ready to give a general definition of an answer set. A set X of atoms is an *answer set* for a program Π if X is minimal among the sets of atoms that satisfy the reduct Π^X [Lifschitz *et al.*, 1999].

In this definition, minimality is understood as minimality relative to set inclusion. In other words, we say that X is an answer set for Π if Π^X is satisfied by X but is not satisfied by any proper subset of X. When applied to a program consisting of rules of form (2), the general definition of an answer set is equivalent to the definition given in Section 1.

Exercise 2.1 Find the answer sets for each of the programs

(a)

$$p; q$$

(b)

$$\begin{array}{l} p; q \\ p \leftarrow q \end{array}$$

(c)

$$\begin{array}{l} p; q \\ p \leftarrow q \\ q \leftarrow p \end{array}$$

Exercise 2.2 Check that each of the one-rule programs

$$p; \, not \;\, p \tag{7}$$

and

$$p \leftarrow not \;\, not \;\, p \tag{8}$$

has two answer sets: $\emptyset$ and $\{p\}$.

The last two examples show that the anti-chain property (Section 1) does not hold, generally, if the rules of a program are not assumed to have form (2). The other property of answer sets mentioned there can be extended to arbitrary programs in the following form: any atom that belongs to an answer set for Π occurs in the head of one of the rules of Π.

Intuitively, rule (7) says: you are free to choose whether or not to include p in the answer set. More generally, the one-rule program

$$(p_1; not \;\, p_1), \ldots, (p_n; not \;\, p_n) \tag{9}$$

(n is a positive integer) has 2^n answer sets—arbitrary sets formed from the atoms $p_1, \ldots, p_n$. Formulas (7) and (9) are examples of *choice expressions*—formulas of the form

$$(F_1; not \;\, F_1), \ldots, (F_n; not \;\, F_n).$$

We will write this formula as

$$\{F_1, \ldots, F_n\}^c. \tag{10}$$

For instance, rule (7) can be abbreviated as $\{p\}^c$, and (9) can be written as $\{p_1, \ldots, p_n\}^c$.

We can prove, that, for any program Π, the program obtained from Π by adding rule (7) has the same answer sets as the result of adding to Π rule (8). The fact that (7) and (8) have the same answer sets is a special case of this theorem corresponding to the empty Π.

The following terminology is useful. We say that a program Π_1 is *weakly equivalent* to a program Π_2 if Π_1 and Π_2 have the same answer sets. Program Π_1 is *strongly equivalent* to Π_2 if, for any program Π, $\Pi_1 \cup \Pi$ is weakly equivalent to $\Pi_2 \cup \Pi$. We can say, for instance, that (7) is strongly equivalent to (8), but these programs are not even weakly equivalent to any program of the kind discussed in Section 1.

Generally, weak equivalence does not imply strong equivalence. For instance, the program

$$p \leftarrow not\ q \tag{11}$$

is weakly equivalent to the program

$$p \leftarrow not\ r \tag{12}$$

because $\{p\}$ is the only answer set for each of them. But (11) is not strongly equivalent to (12). Indeed, if we add q to (11) as the second rule, we'll get a program with the answer set $\{q\}$; if we add the same rule to (12), the result will be a program with the answer set $\{p, q\}$.

The study of strong equivalence tells us how we can simplify a part of a logic program without looking at the rest of it. Methods for checking strong equivalence are discussed in [Lifschitz *et al.*, 2001] and [Turner, 2003].

Exercise 2.3 Show that program (4) is weakly equivalent to the one-rule program

$$p; q$$

but is not strongly equivalent to it.

We know, actually, that no program consisting of rules of form (2) is strongly equivalent to $p; q$ [Turner, 2003].

3 Constraints and Cardinality Expressions

A *constraint* is a rule with the head $\bot$, that is to say, a rule of the form $\bot \leftarrow F$. Such a rule can be abbreviated as $\leftarrow F$. The following theorem (proved, in a slightly different form, in [Lifschitz *et al.*, 1999]) tells us how adding a constraint to a program affects the collection of its answer sets.

Theorem on Constraints *For any program Π and formula F, a set X of atoms is an answer set for $\Pi \cup \{\leftarrow F\}$ iff X is an answer set for Π and does not satisfy F.*

This theorem shows, for instance, that the answer sets for the program

$$\begin{array}{l} \{p,q\}^c \\ \leftarrow p,q \end{array} \tag{13}$$

are the subsets of $\{p, q\}$ that do not satisfy the conjunction p, q:

$$\emptyset, \{p\}, \{q\}.$$

For any formula F formed from the atoms $p_1, \ldots, p_n$, the answer sets for the program

$$\begin{array}{l} \{p_1,\ldots,p_n\}^c \\ \leftarrow \mathit{not}\ F \end{array} \tag{14}$$

are the sets of atoms that satisfy F. This fact gives a reduction of the satisfiability problem for propositional formulas to the problem of computing answer sets. There is also a simpler reduction: program (14) turns out to be strongly equivalent to the one-rule program

$$\{p_1, \ldots, p_n\}^c, F.$$

The art of answer set programming is based on the possibility of representing the collection of sets that we are interested in as the collection of answer sets for a logic program. This is often achieved by combining choice expressions with constraints, as in (14). A program consisting of choice expressions may provide "an approximation from above" for the collection of sets that we want to describe. Constraints may allow us to remove the redundant sets.

For any formulas $F_1, \ldots, F_n$ $(n \geq 0)$ and any nonnegative integer l ("lower bound"), by

$$l \leq \{F_1, \ldots, F_n\} \tag{15}$$

we denote the disjunction of all conjunctions of the form

$$F_{i_1}, \ldots, F_{i_l} \quad (1 \leq i_1 < \cdots < i_l \leq n).$$

For instance,

$$2 \leq \{p, q, r\}$$

stands for

$$(p, q); (p, r); (q, r).$$

By

$$\{F_1, \ldots, F_n\} \leq u \tag{16}$$

where u is a nonnegative integer ("upper bound"), we denote the formula

$$not\ (u + 1 \leq \{F_1, \ldots, F_n\}).$$

A conjunction of the form

$$(l \leq \{F_1, \ldots, F_n\}), (\{F_1, \ldots, F_n\} \leq u)$$

can be abbreviated as

$$l \leq \{F_1, \ldots, F_n\} \leq u. \tag{17}$$

Formulas of forms (15)—(17) will be called *cardinality expressions*. To illustrate the use of such formulas, let us construct first a logic program whose answer sets are the subsets of $\{p_1, \ldots, p_n\}$ that have at least l elements. One way to do this is to combine a choice rule with a constraint, as follows:

$$\begin{array}{l} \{p_1, \ldots, p_n\}^c \\ \leftarrow not\ (l \leq \{p_1, \ldots, p_n\}). \end{array}$$

This combination is strongly equivalent to

$$\{p_1, \ldots, p_n\}^c, l \leq \{p_1, \ldots, p_n\}.$$

Similarly, if we are interested in the subsets of $\{p_1, \ldots, p_n\}$ that have at most u elements, we can write either

$$\begin{array}{l} \{p_1, \ldots, p_n\}^c \\ \leftarrow not\ (\{p_1, \ldots, p_n\} \leq u) \end{array}$$

or, equivalently,

$$\{p_1, \ldots, p_n\}^c, \{p_1, \ldots, p_n\} \leq u.$$

To get all subsets of $\{p_1, \ldots, p_n\}$ with the cardinalities between l and u, we can write

$$\begin{array}{l} \{p_1, \ldots, p_n\}^c \\ \leftarrow not\ (l \leq \{p_1, \ldots, p_n\}) \\ \leftarrow not\ (\{p_1, \ldots, p_n\} \leq u) \end{array}$$

or

$$\{p_1, \ldots, p_n\}^c, l \leq \{p_1, \ldots, p_n\} \leq u.$$

We will abbreviate the conjunctions

$$\begin{array}{c} \{F_1, \ldots, F_n\}^c, l \leq \{F_1, \ldots, F_n\} \\ \{F_1, \ldots, F_n\}^c, \{F_1, \ldots, F_n\} \leq u \\ \{F_1, \ldots, F_n\}^c, l \leq \{F_1, \ldots, F_n\} \leq u \end{array}$$

as

$$\begin{array}{l} l \leq \{F_1, \ldots, F_n\}^c \\ \{F_1, \ldots, F_n\}^c \leq u \\ l \leq \{F_1, \ldots, F_n\}^c \leq u \end{array} \tag{18}$$

respectively.

Exercise 3.1 Write a one-rule program with the answer sets

$$\{p\},\ \{q\},\ \{r\}.$$

Cardinality expressions were introduced in [Niemelä and Simons, 2000] as a primitive construct. Our presentation is based on the possibility of treating them as abbreviations, established in [Ferraris and Lifschitz, 2004].

4 Computing Answer Sets with LPARSE and SMODELS

In this section we discuss the use of the answer set solver created by Ilkka Niemelä's group at the Helsinki University of Technology. The system consists of two programs, LPARSE and SMODELS; the former is a preprocessor. It can be downloaded from its web site,

`http://www.tcs.hut.fi/Software/smodels/` .

The syntax of the input language of LPARSE is more limited in some ways than the class of logic programs defined in Section 1, but it includes many useful special cases. In that language, the head of a rule can be

- an atom,
- empty (that is, $\bot$),
- an expression of one of the forms (10), (18), provided that the formulas F_i are atoms.

In the last case, the superscript ^{c} and the sign $\leq$ are dropped. For instance,

$$\mathtt{1\{p,q\}} \tag{19}$$

in the head of a rule stands for

$$1 \leq \{p, q\}^c. \tag{20}$$

The body can be a conjunction (possibly empty) of

- atoms possibly preceded with *not*,
- expressions of the forms (15)–(17), provided that each F_i is an atom possibly preceded with *not*.

In the last case, the sign $\leq$ is dropped. For instance, expression (19), when it occurs in a body of an LPARSE rule, stands for

$$1 \leq \{p, q\}.$$

Note that this expression is different from (20).

In the input language of LPARSE, as in Prolog, `:-` stands for $\leftarrow$, and each rule is followed by a period.

If we want to find, for instance, the answer sets for the program from Exercise 1.3, we create the file

```
p :- not q.
q :- not p.
r :- p.
r :- q.
```

called, say, `p1.3` . Then we invoke LPARSE and SMODELS as follows:

```
% lparse p1.3 | smodels 0
```

The zero at the end indicates that we want to compute all answer sets; a positive number k would tell SMODELS to terminate after computing k answer sets. The default value of k is 1. The main part of the output generated in response to this command line is the list of the program's answer sets:

```
Answer: 1
Stable Model: r p
Answer: 2
Stable Model: r q
```

(recall that "stable model" means "answer set").

Exercise 4.1 Use SMODELS to find the answer sets for program (13).

Exercise 4.2 Although the syntax of LPARSE does not allow us, generally, to use nested negations, the formula *not not* p can be written as a cardinality expression that LPARSE understands. Find such an expression and use SMODELS to find the answer sets for program (8).

A group of rules that follow a pattern can be often described in a compact way using variables. As in Prolog, variables must be capitalized. Consider, for instance, the programs Π_n from Exercise 1.2. To describe Π_7, we don't have to write out each of its 7 rules. Let's agree to use the symbol `index` to represent a number between 1 and 7. We can write Π_7 as

```
index(1..7).
p(I) :- not p(I+1), index(I).
```

The auxiliary symbols used in the input language of LPARSE to describe the ranges of variables, such as `index`, are called domain predicates. The "definitions" of domain predicates, such as `index(1..7)`, tell LPARSE how to substitute specific values for variables in schematic expressions, such as `p(I) :- not p(I+1)`. Grounding—translating schematic expressions into sets of rules—is the main computational task performed by LPARSE.

When the input file uses domain predicates, the output of SMODELS lists the objects satisfying each domain predicate along with the elements of the answer set:

```
Answer: 1
Stable Model: p(1) p(3) p(5) p(7) index(1) index(2)
index(3) index(4) index(5) index(6) index(7)
```

Information on the extents of domain predicates in the output can be suppressed by including a `hide` directive in the input file:

```
hide index(_).
```

The family of programs Π_n can be described by a schema with the parameter n:

```
index(1..n).
p(I) :- not p(I+1), index(I).
```

When this schematic description is given to LPARSE as input, the value of the constant n should be specified in the command line using the option `-c`, as follows:

```
% lparse -c n=7 p1.2a | smodels 0
```

Exercise 4.3 Consider the program obtained from Π_n by adding the rule

$$p_{n+1} \leftarrow not\ p_1.$$

How many answer sets does this program have, in your opinion? Check your conjecture for $n = 7$ and $n = 8$ using SMODELS.

Variables can be also used "locally" to describe the list of formulas in a cardinality expression. The domain predicate characterizing the range of a local variable is placed inside the braces. For instance, the rule

$$1 \leq \{p_1, \ldots, p_n\} \leq 1$$

can be represented in an LPARSE input file by the lines

```
index(1..n).
1{p(I) : index(I)}1.
```

Exercise 4.4 Consider the program

$$1 \leq \{p_{i1}, \ldots, p_{in}\}^c \leq 1 \qquad (1 \leq i \leq n),$$

where n is a positive integer. How many answer sets does this program have, in your opinion? Check your conjecture for $n = 3$ using SMODELS.

A detailed description of the input language of LPARSE, as well as the complete list of options of LPARSE and SMODELS, can be found in the system's manual, available at its web site.

Besides SMODELS, there are currently two other answer set solvers that use LPARSE for grounding: ASSAT[2] and CMODELS[3]. Among other existing systems for computing answer sets, DLV[4] stands out by virtue of two things: it uses a very efficient grounding mechanism, and it can process rules containing disjunctions of several atoms in the head, as in Exercise 2.1.

5 Answer Set Programming

To solve a problem using ASP means to write a logic program whose answer sets correspond to solutions, and then find a solution using an answer set solver, such as SMODELS. The basic approach to writing such a program is known as the "generate-and-test" strategy. First we write a group of rules whose heads contain choice expressions so that the answer sets for this group of rules correspond to "potential solutions"—an easy-to-describe superset of

[2] `http://assat.cs.ust.hk/` .
[3] `http://www.cs.utexas.edu/users/tag/cmodels/` .
[4] `http://www.dbai.tuwien.ac.at/proj/dlv/` .

the set of solutions. Then we add a group of constraints that weed out the potential solutions that are not solutions.

Consider, for instance, the use of this method to solve the n-queens problem. The goal is to place n queens on an $n \times n$ chessboard so that no two queens would be placed on the same row, column or diagonal. A solution can be described by a set of atoms of the form $q(i,j)$ $(1 \leq i,j \leq n)$; including $q(i,j)$ in the set indicates that there is a queen at position (i,j). A solution is a set X satisfying the following conditions:

1. The cardinality of X is n.
2. X does not contain a pair of different atoms of the form $q(i,j)$, $q(i',j)$ (two queens in the same row).
3. X does not contain a pair of different atoms of the form $q(i,j)$, $q(i,j')$ (two queens in the same column).
4. X does not contain a pair of different atoms $q(i,j)$, $q(i',j')$ with $|i'-i| = |j'-j|$ (two queens on the same diagonal).

The sets satisfying Conditions 1 and 2 can be described by rules similar to the program from Exercise 4.4:

$$1 \leq \{q(1,j), \dots, q(n,j)\}^c \leq 1 \qquad (1 \leq j \leq n)$$

(exactly one queen in each row). These rules form the "generate" part of our program. The "test" part consists of the constraints expressing Condition 3

$$\leftarrow q(i,j), q(i,j') \qquad (1 \leq i,j,j' \leq n;\ j < j')$$

and Condition 4

$$\leftarrow q(i,j), q(i',j') \qquad (1 \leq i,i'j,j' \leq n;\ j < j';\ |i'-i| = j'-j).$$

Here is a representation of this program in the input language of LPARSE:

```
number(1..n).

1{q(I,J) : number(I)}1 :- number(J).

:- q(I,J), q(I,J1),
   number(I;J;J1), J<J1.

:- q(I,J), q(I1,J1),
   number(I;I1;J;J1), J<J1, abs(I1-I)==J1-J.
hide number(I).
```

The expression `number(I;J;J1)` is the LPARSE abbreviation for

```
number(I), number(J), number(J1).
```

Note the conditions `J<J1` and `abs(I1-I)==J1-J` in the bodies of rules. By including them, we tell LPARSE how to limit the choice of values used in the process of grounding.

Exercise 5.1 Use SMODELS to find all solutions to the 8 queens problem that (a) have a queen at (1,1); (b) have no queens in the 4×4 square in the middle of the board.

We will show now how to use ASP to estimate the size of Schur numbers. A set A of integers is called *sum-free* if there are no numbers x, y in A such that $x + y$ is in A also (x and y do not need to be different). The *Schur number* $S(k)$ is the largest integer n for which the interval $\{1, \ldots, n\}$ can be partitioned into k sum-free sets.[5]

For specific values of k, $S(k)$ can be computed (or estimated) using a program whose answer sets correspond to the partitions of $\{1, \ldots, n\}$ into k (possibly empty) sum-free sets. We will use the atoms $a_i(x)$ ($1 \leq i \leq k$, $1 \leq x \leq n$) to express that x belongs to the i-th set A_i. The choice rules

$$1 \leq \{a_1(x), \ldots, a_k(x)\} \leq 1 \qquad (1 \leq x \leq n)$$

describe "potential solutions"—partitions of $\{1, \ldots, n\}$ into k sets. The constraints

$$\leftarrow a_i(x), a_i(y), a_i(x+y) \qquad (x, y \geq 1,\ x + y \leq n,\ 1 \leq i \leq k)$$

eliminate the sets A_i that are not sum-free.

Here is how this program can be written in the input language of LPARSE:

```
number(1..n).
subset(1..k).

1{a(I,X) : subset(I)}1 :- number(X).

:- a(I,X), a(I,Y), a(I,X+Y),
   subset(I), number(X;Y), X+Y<=n.

hide number(_), subset(_).
```

[5] See `http://mathworld.wolfram.com/SchurNumber.html` .

Exercise 5.2 Use this program to find $S(3)$.

The number of answer sets for the ASP program above is greater than the number of partitions of $\{1,\ldots,n\}$ into k sum-free sets, because the sets in any such partition can be arranged in a sequence $A_1,\ldots,A_k$ in many possible ways. In particular, the subsets in a partition can be always ordered in such a way that

$$\begin{array}{l} 1 \in A_1, \\ 2 \in A_1 \cup A_2, \\ \ldots \\ k \in A_1 \cup \cdots \cup A_k. \end{array}$$

Consequently, if we add the constraint

```
:- not a(I,X), subset(I), number(X), X <=I.
```

to the program above, the property of the program that we are interested in will be preserved—it will have an answer set iff $\{1,\ldots,n\}$ can be partitioned into k sum-free sets. Adding this "symmetry breaking" constraint will actually improve the computation time of SMODELS for many values of k and n, because, generally, SMODELS is good at exploiting constraints for restricting search.

Exercise 5.3 Investigate the effect of this optimization on the computation time of SMODELS for $k = 3$, $n = 14$ and for $k = 4$, $n = 38$.

In many cases, the constraints in the test part of the program use auxiliary atoms, defined in terms of the atoms occurring in the generate part. The definitions of the auxiliary atoms form then a third component of the program, besides the "generate" and "test" parts—its "define" part.

Consider, for instance, the problem of finding a Hamiltonian circuit in a given graph—a closed path that visits every node exactly once. A Hamiltonian circuit in a graph G can be thought of as a subgraph C of G that has the same set V of vertices as G and satisfies two conditions:

1. Every vertex is adjacent in C to exactly two vertices.

2. Every vertex is reachable in C from some fixed vertex v_0.

The program below represents an edge $\{u,v\}$ of G by the atom $in(u,v)$, where $u < v$ ($<$ is a fixed total order on V). The presence of this atom in an answer set indicates that the edge $\{u,v\}$ is included in C. The generate part of the program consists of the rules

$$\{in(u,v)\}^c \tag{21}$$

for all edges $\{u, v\}$ of G $(u < v)$. Thus any set of edges is a potential solution. To encode the two conditions that characterize Hamiltonian circuits, we introduce the auxiliary atoms $adj(u, v)$ $(u < v)$ that represent adjacency in C. These atoms are "defined" in terms of $in(u, v)$ by the rules

$$adj(u, v), adj(v, u) \leftarrow in(u, v) \tag{22}$$

for all edges $\{u, v\}$ of G $(u < v)$. Then Condition 1 above can be expressed by the constraints

$$\begin{array}{l} \leftarrow \{adj(u, v) \ : \ v \in V\} \leq 1 \\ \leftarrow 3 \leq \{adj(u, v) \ : \ v \in V\} \end{array} \tag{23}$$

for all $u \in V$. For Condition 2, we introduce the auxiliary atoms $r(u)$ that express the reachability of u from v_0. They are "defined recursively" by the rules

$$\begin{array}{l} r(v_0) \\ r(u) \leftarrow r(v), adj(u, v) \end{array} \tag{24}$$

for all edges $\{u, v\}$ of G. Using these atoms, we express Condition 2 by the constraints

$$\leftarrow not\ r(u) \tag{25}$$

for all $u \in V$.

Answer sets for program (21)–(25) are in a 1–1 correspondence with the Hamiltonian circuits in G. Rules (22) and (24) form the define part of the program, and rules (23) and (25) form the test part.

The following file `hc` is an LPARSE encoding of program (21)–(25), in which we assume that the vertices of G are integers, and $v_0 = 0$:

```
{in(U,V)} :- edge(U,V).

adj(U,V) :- in(U,V), vertex(U;V).
adj(V,U) :- in(U,V), vertex(U;V).

:- {adj(U,V) : vertex(V)}1, vertex(U).
:- 3{adj(U,V) : vertex(V)}, vertex(U).

r(0).
r(U) :- r(V), adj(U,V), vertex(U;V).

:- not r(U), vertex(U).

hide.
show in(_,_).
```

(The last two lines tell SMODELS to display the `in` atoms only.)

This file needs to be appended to a description of G in the form of a definition of the domain predicates `vertex` and `edge`. For instance, the file

```
vertex(0..7).

edge(U,U+1) :- vertex(U), U!=3, U!=7.
edge(0,3).
edge(4,7).
edge(U,U+4) :- vertex(U), U<4.
```

describes the vertices and edges of a cube. Assuming that this file is called `cube`, SMODELS can be instructed to find a Hamiltonian circuit by the command

```
% lparse cube hc | smodels
```

Exercise 5.4 The vertices of graph G_n are the points (x, y) with coordinates $x, y \in \{0, \dots, n-1\}$; two vertices are adjacent if the distance between them is 1. Use SMODELS to find out whether G_5 has a Hamiltonian circuit.

Syntactically, rules in the generate and test parts of an ASP program are somewhat different from Prolog rules: Prolog has neither choice expressions nor constraints. On the other hand, the define part of an ASP program does look similar to a Prolog program, and writing define rules in ASP is based on the same intuition as writing Prolog programs.

For "generate-and-test" programs, such as the first two examples above, proving correctness is usually not difficult. To prove the correctness of such a program, we often need no theory of answer sets beyond the basic properties of choice expressions (to describe the answer sets for the generate part) and the theorem on constraints (to describe the effect of adding the test part). If, on the other hand, a define part is present, as in the Hamiltonian circuit example, more complex mathematical tools are required for a correctness proof. This is discussed in [Erdoğan and Lifschitz, 2004], where the idea of splitting [Lifschitz and Turner, 1994] is used to describe the effect of adding the define part to the generate part.

In each of the following exercises, show how to solve the given computational problem using SMODELS. As in the Hamiltonian circuit example, use specific data (of your choice) to test your program.

Exercise 5.5 A graph G is said to be *n-colorable* if there exists a function f from its vertices to numbers $1, \dots, n$ such that $f(x) \neq f(y)$ for every pair of adjacent vertices x, y. Determine whether G is n-colorable.

Exercise 5.6 A *clique* in a graph G is a subset of its vertices whose elements are pairwise adjacent. Determine whether G has a clique of cardinality n.

Exercise 5.7 You are organizing a large New Year's Eve party. There will be n tables in the room, with m chairs around each table. You need to select a table for each of the guests, who are assigned numbers from 1 to mn, so that two conditions are satisfied. First, some guests like each other and want to sit together; accordingly, you are given a set A of two-element subsets of $\{1, \dots, mn\}$, and, for every $\{i, j\}$ in A, guests i and j should be assigned the same table. Second, some guests dislike each other and want to sit at different tables; accordingly, you are given a set B of two-element subsets of $\{1, \dots, mn\}$, and, for every $\{i, j\}$ in B, guests i and j should be assigned different tables. The goal is to find such a seating arrangement or determine that this is impossible.

Exercise 5.8 You are in charge of assigning referees to the papers submitted to a conference. Each of the n submissions needs to be assigned to a few referees from among the m members of the Program Committee. The PC members have read the abstracts of all submissions, and each of them gave you the numbers of the submissions that he is qualified to referee; let A_i $(1 \leq i \leq m)$ be the subset of $\{1, \dots, n\}$ given to you by the i-th committee member. You need to select, for each i, a set X_i of papers to be assigned to the i-th committee member so that three conditions are satisfied. First, each X_i should be a subset of A_i. Second, the cardinality of each X_i should be between a lower bound l and an upper bound u. Third, each paper should be assigned to exactly k referees. The goal is to find such an assignment of papers to referees or determine that this is impossible.

Exercise 5.9 A *wire routing problem* is determined by a list

$$\{v_1, v_1'\}, \dots, \{v_k, v_k'\}$$

of disjoint pairs of nodes of graph G_n (see Exercise 5.4). A *solution* to the problem is a list of paths $\alpha_1, \dots, \alpha_k$ in G_n such that

- α_i connects v_i with v_i' $(i = 1, \dots, k)$, and
- no two paths on the list pass through a common node.

Given a wire routing problem, the goal is to find a solution such that the length of each path in it does not exceed a given upper bound u, or determine that this is impossible.

6 How SMODELS Computes Answer Sets

We will describe now, in a simplified form, the algorithm that SMODELS uses for computing answer sets. This discussion is limited to the case when the rules of the input program have the form (2) and assumes that we are interested in finding one answer set only.

As we know from Section 1, if every rule of a program Π has form (1) then Π has a unique answer set. We will denote this set by $as(\Pi)$.

The operation of SMODELS is based on the following theorem,[6] where Π stands for a program consisting of rules of form (2).

Theorem on Approximating Answer Sets *Let X be an answer set for Π. For any sets L, U of atoms, if*

$$L \subseteq X \subseteq U$$

then

$$L \cup as(\Pi^U) \subseteq X \subseteq U \cap as(\Pi^L).$$

This theorem tells us that if we want to find the answer sets for Π between a lower bound L and an upper bound U then these bounds can be generally made more restrictive without losing solutions.

The process of replacing the lower bound L by $L \cup as(\Pi^U)$ and the upper bound U by $U \cap as(\Pi^L)$ can be iterated, as in the algorithm EXPAND shown in Figure 1. After a call to EXPAND, L can only become larger, and U can only become smaller; but every answer set for Π that satisfied the condition $L \subseteq X \subseteq U$ before the call will satisfy that condition after the call as well.

If a call to EXPAND produces a pair of sets L, U such that L is not a subset of U then we can conclude that Π has no answer sets between the initially given bounds.

On the other hand, if $L = U$ after a call to EXPAND then L is an answer set for Π. Indeed, according to the termination condition of the loop, $L = L'$ and $U = U'$, which means that

$$L = L \cup as(\Pi^U)$$

and

$$U = U \cap as(\Pi^L).$$

[6]The actual SMODELS algorithm is more sophisticated and uses several propagation methods not covered by our description.

```
EXPAND(L, U)
    repeat
        L' := L;
        L := L ∪ as(Π^U);
        if L ⊈ U then return;
        U' := U;
        U := U ∩ as(Π^L);
        if L ⊈ U then return
    until L = L' and U = U'
```

Figure 1: Approximating an answer set

Consequently

$$as(\Pi^U) \subseteq L$$

and

$$U \subseteq as(\Pi^L).$$

In combination with the assumption $L = U$, these formulas imply that $as(\Pi^L) = L$. According to the definition of an answer set from Section 1, the last formula expresses that L is an answer set for Π.

In some cases, this property of EXPAND allows us to find an answer set simply by executing EXPAND with the empty set as L and the set of all atoms as U. This method is applicable only when the given program has a unique answer set.

Exercise 6.1 Use EXPAND to find an answer set for program (3).

The algorithm SMODELS (Figure 2) computes an answer set between L and U, or determines that the program does not have such an answer set. SMODELS begins by calling EXPAND. Then, if necessary, it selects an atom A and calls itself recursively, first to try to find an answer set between $L \cup \{A\}$ and U, and then to try to find an answer set between L and $U \setminus \{A\}$. Initially SMODELS is invoked with the empty set as L and the set of all atoms as U.

Exercise 6.2 Apply the algorithm SMODELS to programs (4) and (5).

In its structure, SMODELS is similar to the Davis-Putnam-Logemann-Loveland procedure [Davis *et al.*, 1962] that serves as the basis of many fast propositional satisfiability solvers. First it tries to find an answer set by calling EXPAND, just as DPLL begins by trying to find a satisfying truth assignment for the given set of clauses by unit clause propagation. If that

```
SMODELS(L, U)
    EXPAND(L, U);
    if L ⊈ U then return;
    if L = U then exit with L;
    A := an atom from U \ L;
    SMODELS(L ∪ {A}, U);
    SMODELS(L, U \ {A})
```

Figure 2: Computing an answer set

attempt does not succeed, each procedure "branches" and calls itself recursively.

The problem of computing answer sets when the rules have form (2) or other forms permitted by the input language of LPARSE is easier than the general case. Verifying the existence of an answer set for such programs is *NP*-complete [Marek and Truszczyński, 1991]. If disjunctions of atoms are allowed in the heads of rules, as in the input language of DLV, the problem becomes Σ_2^P-complete [Eiter and Gottlob, 1993].

7 External Support, Loops and Tight Programs

Two of the answer set solvers mentioned in Section 4, ASSAT and CMODELS, operate by reducing the problem of computing answer sets to the satisfiability problem for propositional formulas and by invoking a SAT solver to do search. The translation they use is more complicated than the opposite reduction (14) of propositional satisfiability to ASP. It is based on the idea of "external support," and we will discuss it here for the case of rules of form (2).

Consider a program Π consisting of rules of form (2). For any set Z of atoms, the *external support formula* for Z is the disjunction of the bodies

$$A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n$$

of all rules (2) of Π such that

$$A_0 \in Z,\ A_1, \ldots, A_m \notin Z. \tag{26}$$

Intuitively, a rule satisfying (26) "supports" an element of Z (its head A_0 is in Z) by "external evidence" (the elements $A_1, \ldots, A_m$ of the body are not in Z). We will denote the external support formula by $ES_{\Pi,Z}$.

In the following theorem, Π stands for a program consisting of rules of form (2).

Theorem on External Support *A set X of atoms is an answer set for Π iff*

- *X satisfies Π, and*
- *for every set Z of atoms, X satisfies the formula*

$$(\textstyle\bigvee_{A\in Z} A) \supset ES_{\Pi,Z}. \tag{27}$$

This theorem is related to the concept of an assumption set from [Saccá and Zaniolo, 1990] and is due to Joohyung Lee [2004].

Take, for instance, the program

$$\begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \\ r \leftarrow p \\ r \leftarrow q \end{array} \tag{28}$$

familiar to us from Exercise 1.3. According to the theorem on external support, a subset X of $\{p,q,r\}$ is an answer set for this program iff it satisfies, first, the rules of the program, that is, the formulas

$$\begin{array}{rcl} \neg q & \supset & p \\ \neg p & \supset & q \\ p & \supset & r \\ q & \supset & r \end{array} \tag{29}$$

and, second, all formulas of form (27), that is,

$$\begin{array}{rcl} \top & \supset & \top \\ p & \supset & \neg q \\ q & \supset & \neg p \\ r & \supset & p \vee q \\ p \vee q & \supset & \neg p \vee \neg q \\ p \vee r & \supset & \neg q \vee q \\ q \vee r & \supset & \neg p \vee p \\ p \vee q \vee r & \supset & \neg p \vee \neg q. \end{array} \tag{30}$$

The theorem on external support provides a reduction of the problem of computing answer sets to SAT. But it cannot be used directly as a basis for the design of an answer set solver, because the number of formulas (27) grows exponentially with the number of atoms. Fortunately, the statement of the theorem remains true even if we limit the choice of Z to sets of a special form, called "loops."

To define the concept of a loop, we need first to introduce a binary relation on the set of atoms associated with the given program Π. This relation R_Π is defined as follows: $A \, R_\Pi \, A'$ if Π contains a rule (2) such that $A = A_0$ and $A' \in \{A_1, \dots, A_m\}$. For instance, if Π is program (28) then $r R_\Pi p$ and $r R_\Pi q$; for all other pairs of atoms R_Π does not hold. A *loop* in Π is a nonempty set Z of atoms such that for any $A, A' \in Z$ there exists a finite sequence $A_1, \dots, A_k$ ($k \geq 1$) of atoms from Z satisfying the conditions

$$A = A_1, \; A' = A_k, \; A_i \, R_\Pi \, A_{i+1} \; (1 \leq i < k).$$

It is clear that, for any atom A, $\{A\}$ is a loop. Program (28) has no loops other than the singletons $\{p\}$, $\{q\}$, $\{r\}$. The program

$$\begin{array}{l} p \leftarrow \mathit{not}\ q \\ q \leftarrow r \\ r \leftarrow q \end{array} \tag{31}$$

has a nonsingleton loop, $\{q, r\}$.

Theorem on Loops *A set X of atoms is an answer set for Π iff*

- *X satisfies Π, and*
- *for every loop Z of Π, X satisfies the formula*

$$\left(\bigvee_{A \in Z} A\right) \supset ES_{\Pi,Z}.$$

The concept of a loop and the theorem on loops are due to Fangzhen Lin and Yuting Zhao [2004], the creators of ASSAT. Our presentation follows [Lee, 2004].

The theorem on loops shows, for instance, that the answer sets for (28) can be characterized as the sets satisfying (29) and the formulas in lines 2–4 of (30). In application to program (31), the theorem on loops asserts that its answer sets are the sets satisfying the formulas

$$\begin{array}{rcl} \neg q & \supset & p \\ r & \supset & q \\ q & \supset & r \end{array}$$

and

$$\begin{array}{rcl} p & \supset & \neg q \\ q & \supset & r \\ r & \supset & q \\ q \vee r & \supset & \bot . \end{array}$$

Some programs have exponentially many loops. Complexity considerations show that exponential growth in size is likely to be unavoidable whenever logic programs are mapped into equivalent propositional formulas [Lifschitz and Razborov, 2004]. On the other hand, in many cases of interest to ASP the number of loops has about the same order of magnitude as the number of atoms.

Exercise 7.1 (a) Use the theorem on loops to find the answer sets for the programs from Exercise 4.3. (b) If you erase all negations in those programs, how will that affect your calculation?

If Z is a singleton, formula (27) is closely related to the concept of program completion introduced in [Clark, 1978]. To be more precise, take any program Π consisting of rules (2) such that A_0 is different from the atoms $A_1, \dots, A_m$ (we will call such rules *nontrivial*). The conjunction of the rules of Π with formulas (27) for all singleton sets Z is equivalent to the completion of Π.

The process of computing answer sets using loop formulas is particularly simple when the program we are interested in has no loops other than singletons. Such programs are called *tight*. (This is a little less restrictive than the usual definition of tightness, according to which a trivial rule in the program makes it nontight.) The fact that answer sets for a tight program can be characterized as the models of its completion was discovered by François Fages [1991], and using this fact for computing answer sets was proposed in [Babovich *et al.*, 2000]. Tightness and loops are extended to rules more general than (2) in [Erdem and Lifschitz, 2003, Lee and Lifschitz, 2003].

8 Conclusion

ASP is an increasingly popular approach to solving search problems. Dissertations have been written about ASP [Simons, 2000, Erdem, 2002], and a monograph on ASP has been published [Baral, 2003] (although the term "answer set programming" is not used in it). The European Working Group on Answer Set Programming[7] has nodes in 17 universities.

Combinatorial search has been at the center of attention of many mathematicians, computer scientists and specialists in operations research and artificial intelligence for decades. A good programming methodology is expected to allow us to write programs that are not only efficient, but also easy to understand and modify. The future will show whether ASP can make a significant contribution to this direction of research.

Acknowledgements

These notes are based on the graduate seminar that I have taught twice at the University of Texas, and I am grateful to my students for their feedback. Many thanks also to Chitta Baral, Esra Erdem, Selim Erdoğan, Paolo Ferraris, Ilkka Niemelä, Joohyung Lee, Yuliya Lierler, Victor Marek and Miroslaw Truszczyński for comments on an early draft. This work was partially supported by the Texas Higher Education Coordinating Board under Grant 003658-0322-2001.

References

[Apt *et al.*, 1988] Krzysztof Apt, Howard Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, San Mateo, CA, 1988.

[Babovich *et al.*, 2000] Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming.[8] In *Proceedings of International Workshop on Nonmonotonic Reasoning (NMR)*, 2000.

[Baral, 2003] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

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

[Davis *et al.*, 1962] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[7] `http://wasp.unime.it/` .

[8] `http://arxiv.org/abs/cs.ai/0003042` .

[Eiter and Gottlob, 1993] Thomas Eiter and Georg Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In Dale Miller, editor, *Proceedings of International Logic Programming Symposium (ILPS)*, pages 266–278, 1993.

[Erdem and Lifschitz, 2003] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[Erdem, 2002] Esra Erdem. *Theory and Applications of Answer Set Programming*. PhD thesis, University of Texas at Austin, 2002.

[Erdoğan and Lifschitz, 2004] Selim Erdoğan and Vladimir Lifschitz. Definitions in answer set programming. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 114–126, 2004.

[Fages, 1991] François Fages. A fixpoint semantics for general logic programs compared with the well–supported and stable model semantics. *New Generation Computing*, 9:425–443, 1991.

[Ferraris and Lifschitz, 2004] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions.[9] *Theory and Practice of Logic Programming*, 2004. To appear.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080, 1988.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Kautz and Selman, 1992] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, pages 359–363, 1992.

[Lee and Lifschitz, 2003] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 451–465, 2003.

[9] `http://www.cs.utexas.edu/users/vl/papers/weight.ps` .

[Lee, 2004] Joohyung Lee. A model-theoretic counterpart of loop formulas. In preparation, 2004.

[Lifschitz and Razborov, 2004] Vladimir Lifschitz and Alexander Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 2004. To appear.

[Lifschitz and Turner, 1994] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proceedings of International Conference on Logic Programming (ICLP)*, pages 23–37, 1994.

[Lifschitz *et al.*, 1999] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[Lifschitz *et al.*, 2001] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.

[Lifschitz, 1999] Vladimir Lifschitz. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.

[Lifschitz, 2002] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

[Lin and Zhao, 2004] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157:115–137, 2004.

[Marek and Truszczyński, 1991] Victor Marek and Mirosław Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.

[Marek and Truszczyński, 1999] Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[Niemelä and Simons, 2000] Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.

[Niemelä, 1999] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

[Reiter, 1980] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Saccá and Zaniolo, 1990] Domenico Saccá and Carlo Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 205–217, 1990.

[Simons, 2000] Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, 2000.

[Turner, 2003] Hudson Turner. Strong equivalence made easy: nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4,5):609–622, 2003.

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

[Van Gelder *et al.*, 1991] Allen Van Gelder, Kenneth Ross, and John Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.