From Felicitous Models to Answer Set Programming

Vladimir Lifschitz

University of Texas at Austin, USA

Abstract. Felicitous models were defined by Kit Fine in 1987 for the purpose of describing the semantics of negation in the programming language Prolog. This is an expository article about that invention, and about the events in the area of computer science that followed. Sophisticated software systems for generating felicitous models have been designed, and they became the basis of a new knowledge representation paradigm, called answer set programming. That methodology is used today for encoding and solving computational problems in many areas of science and technology.

1 Introduction

Felicitous models were defined by Kit Fine [11] for the purpose of describing the semantics of negation in the programming language Prolog. As often happens with good ideas in science, this idea was developed independently, in somewhat different forms, by several other researchers. Fine's paper was presented at the Eighth International Congress of Logic, Methodology and Philosophy of Science, held in 1987; Nicole Bidoit and Christine Froidevaix [4] described their work at the Second Annual IEEE Symposium on Logic in Computer Science the same year; Michael Gelfond's paper [15] was given at the AAAI National Conference on Artificial Intelligence, in 1987 as well. A paper by Gelfond and Lifschitz, initially submitted to the Seventh ACM Symposium on Principles of Database Systems, became "one of the papers which time considerations prevented from being presented at the Symposium"; it was contributed in 1988 to the International Logic Programming Conference and Symposium [17]. The last of these papers became a standard reference, and felicitous models are often called "stable models"—the term introduced in that paper. They are also referred to as answer sets.

In this expository article we discuss the invention of felicitous models (Sections 2–9) and tell the story of surprising developments in the area of computer science that followed (Sections 10–13). Sophisticated software systems for generating felicitous models have been designed, and they became the basis of a new knowledge representation paradigm, called answer set programming [25, 27]. This methodology has been used for encoding and solving computational problems in a variety of areas—from historical linguistics [5] to spacecraft design [29].

2 Prolog

Propositional formulas of the form

$$A_1 \wedge \cdots \wedge A_n \to A_{n+1} \qquad (n \geq 0), \tag{1}$$

where each A_i is an atom, can be rewritten as Prolog rules, and by running a Prolog interpreter we can learn which atoms are entailed by a set of formulas of this form. For instance, the set of formulas

$$\begin{array}{l} p_1, \\ p_1 \to p_2, \\ p_2 \wedge p_3 \to p_4 \end{array} \tag{2}$$

can be represented by the Prolog program

```
p(1).
p(2) :- p(1).
p(4) :- p(2), p(3).
```

Most Prolog rules, like the last two in this example, contain the "if" symbol `:-` and are similar to implications written backwards. The part of a rule to the left of this symbol is called the head of the rule and corresponds to the consequent of the implication; the part to the right, called the body, corresponds to the antecedent. A Prolog system will tell us that the atoms p_1 and p_2 are entailed by formulas (2), and the atoms p_3 and p_4 are not.

The answers given by Prolog in this example can be interpreted also in a different way. In logic programming, it is customary to identify a truth assignment with the set of atoms that get the value *true*. For example, we can represent assigning the value *true* to p_1 and p_2, and the value *false* to all other atoms, by the set $\{p_1, p_2\}$. This set is a model of formulas (2), in the sense that these formulas are satisfied by the corresponding truth assignment. The sets $\{p_1, p_2, p_4\}$ and $\{p_1, p_2, p_3, p_4\}$ are models of (2) as well. The model $\{p_1, p_2\}$ differs from the others in that it is *minimal*—its proper subsets are not models of (2).

We can say that in application to a set of formulas of form (1) Prolog calculates the minimal model of that set [8]. To be precise, Prolog calculates that model if it terminates; sometimes it goes to an infinite loop. That will happen, for example, if we try to use Prolog to find the minimal model of the formulas

$$\begin{array}{l} p_1 \to p_2, \\ p_2 \to p_1. \end{array} \tag{3}$$

The minimal model of a set Π of formulas of form (1) can be described as the result of accumulating the atoms that are necessary for satisfying all formulas in Π. For instance, the empty set (all atoms are false) does not satisfy the first of formulas (2); include p_1. The set $\{p_1\}$ does not satisfy the second formula; add p_2. The set $\{p_1, p_2\}$ satisfies all formulas (2); this is the minimal model. Formulas (3) are satisfied by the empty set, so that the process of accumulating atoms stops at the very beginning.

3 Negation as Failure

The observations in Section 2 apply to Prolog rules that correspond to formulas of form (1). How can we extend them to propositional formulas of the form

$$L_1 \wedge \cdots \wedge L_n \to H \qquad (n \geq 0), \tag{4}$$

where each L_i is a literal (atom or negated atom), and H is an atom? (We denote this atom by H because it corresponds to the head of a Prolog rule.) This is the question that led Kit Fine to the idea of a felicitous model.

Consider, for instance, the formulas

$$\begin{array}{l} p_1, \\ p_1 \to p_2, \\ p_2 \wedge \neg p_3 \to p_4 \end{array} \tag{5}$$

and the corresponding program, in which the Prolog negation symbol `\+` is used to represent $\neg$ in the last line:

```
p(1).
p(2) :- p(1).
p(4) :- p(2), \+ p(3).
```

Formulas (5) have two minimal models:

$$M_1 = \{p_1, p_2, p_3\}, \ M_2 = \{p_1, p_2, p_4\}.$$

The answers to queries given by Prolog correspond to M_2. How can this choice be justified? In what sense is this model better than M_1?

The difference between M_1 and M_2 can be informally explained if we look at formulas (5) as rules for generating atoms. The first line allows us to generate p_1. The implication in the second line allows us to generate p_2 if p_1 has been generated. The last of formulas (5) allows us to generate p_4 if two conditions are satisfied: p_2 has been generated, and any attempt to use all these formulas to generate p_3 would fail. Then we can say: there is no way to use formulas (5) to generate p_3, because the only atoms that these formulas allow us to generate, under various assumptions, are p_1, p_2, and p_4. Consequently generating p_4 using the last of these formulas is justified. The set of atoms that have been generated is what we denoted by M_2. In this sense, M_2 is the "preferred" model of formulas (5).

Explanations in terms of rules for generating atoms, and of the failure of an attempt to use rules, comports with the way Prolog programmers think: they say that their programs consist of rules, and that the symbol `\+` represents "negation as failure" [6]. (And in this article we will sometimes refer to formulas as rules, and to sets of formulas as programs.) But how can we turn such an informal explanation into a mathematical definition?

Digression on intuitionistic logic. If we replace the last of formulas (5) by the equivalent formula

$$p_2 \wedge \neg p_4 \to p_3$$

then model M_1 will be considered preferred. Thus it appears that the property of being preferred is not invariant with respect to equivalent transformations of formulas. It is interesting that the two formulas in this example are equivalent classically, but not intuitionistically. In Section 9 we will say more about the relationship between logic programs and intuitionistic logic.

4 Stratified Programs

Before presenting his own approach to identifying preferred models, Fine discussed other proposals described in the literature and argued that they are not entirely satisfactory. There is, however, a special case—not mentioned in Fine's paper—for which a noncontroversial definition of the preferred model was available: the case of logic programs that are "stratified," or "free from recursive negation" [2, 34].

To stratify a finite set Π of rules of form (4) means to partition it into subsets $\Pi_1, \dots, \Pi_k$ so that if a rule (4) belongs to Π_i $(1 \le i \le k)$ then

(a) whenever a literal L_j in the antecedent of (4) is an atom, every rule in Π with that atom in the consequent belongs to $\Pi_1 \cup \cdots \cup \Pi_i$, and
(b) whenever a literal L_j in the antecedent of (4) is a negative literal $\neg A$, every rule in Π with the consequent A belongs to $\Pi_1 \cup \cdots \cup \Pi_{i-1}$.

For example, every set of rules of form (1) is stratified: include the entire set in one stratum Π_1. Program (5) is stratified using a single stratum as well. The program consisting of the rules

$$\neg p_i \to p_{i+1} \qquad (1 \le i \le k), \tag{6}$$

where k is a positive integer, is also stratified: take $\Pi_i = \{\neg p_i \to p_{i+1}\}$. On the other hand, if Π contains the rule

$$\neg p \to p \tag{7}$$

then Π is not stratified, because that rule would have to belong to each of the two disjoint sets Π_i, $\Pi_1 \cup \cdots \cup \Pi_{i-1}$. Similarly, a program is not stratified if it contains the rules

$$\begin{array}{l} \neg p_1 \to p_2, \\ \neg p_2 \to p_1. \end{array} \tag{8}$$

Constructing the preferred model of a stratified program Π starts with including the consequents of the implications in Π_1 that must be included to satisfy all those implications, as in the example at the end of Section 2. Then the consequents of the implications in Π_2 are added when this is necessary to satisfy the implications in Π_2, and so on. If an atom does not occur in any of the consequents, as p_4 in (5), then it is never included.

For example, constructing the preferred model of program (6) starts with including p_2, to satisfy the formula $\neg p_1 \to p_2$ from the first stratum. Since $\{p_2\}$

satisfies the formula $\neg p_2 \to p_3$ from the second stratum, p_3 is not included. To satisfy $\neg p_3 \to p_4$, we add p_4, and so on, until the stratum Π_k is reached. Thus the preferred model of (6) is $\{p_2, p_4, \ldots\}$.

This process is known as the iterated fixpoint construction. To use it as the definition of the preferred model of a stratified program, one has to show that if this construction is applied to different stratifications of the same program then the preferred model obtained at the end will be the same [2, Theorem 11].

5 Felicitous Models

To characterize preferred models in the general case, Fine writes, "suppose that we make a hypothesis as to which statements are false. Then this hypothesis can be used ... to detach negative statements" from the antecedents of the given implications [11, page 285].

Take, for example, program (5), and consider the hypothesis that p_3 is false. Then the conjunctive term $\neg p_3$ in the last line of (5) is true, and "detaching" it turns (5) into the program

$$\begin{array}{l} p_1, \\ p_1 \to p_2, \\ p_2 \to p_4. \end{array} \tag{9}$$

This is a set of rules of form (1), and its minimal model is

$$\{p_1, p_2, p_4\}. \tag{10}$$

In this example, the atom p_3, which we assumed initially to be false, happens to be the only atom that does not belong to the minimal model (10) that we arrived at. In this sense, the hypothesis that p_3 is false (and that all other atoms occurring in the program are true) was a "happy" hypothesis.

> Instead of thinking in terms of posited falsehoods, we can think in terms of posited truths. A happy hypothesis is then one under which the posited truths coincide with the generated truths. Thus a happy hypothesis is, in a certain sense, self-verifying. It is *verifying*, since what one takes to be the truth turns out to be the truth; and it is *self*-verifying, since it is partly because what one takes the truth to be what it is that it is what it is [11, page 286].

A model is *felicitous* if it "embodies a happy hypothesis" and thus "leads to the whole truth and nothing but the truth."

Let us go back to example (5) and consider now the hypothesis that p_3 is not among the false atoms. Then the antecedent in last line of (5) is false, and that rule can be disregarded. The minimal model of the remaining two rules is $\{p_1, p_2\}$. A situation when "some statement is neither a posited falsehood nor a generated truth," as p_3 in this example, indicates a "gap"; a hypothesis leading to a gap is not happy. In other cases, we may find that "some statement is both a posited falsehood and a generated truth"; this is a "glut"—another kind of

evidence that the hypothesis is not happy. A happy hypothesis "leads to neither gaps not gluts."

In the case of program (5), set (10) is the only felicitous model—the only set of atoms that leads to neither gaps not gluts. Rule (7) has no felicitous models, and program (8) has two: $\{p_1\}$ and $\{p_2\}$.

The concept of a felicitous model describes the behavior of Prolog in the following sense: *if Π is a program consisting of rules of form (4) that has a unique stable model, and Prolog terminates when applied to Π, then its output describes the felicitous model of Π.* In application to a program that has no felicitous models, such as (7), or several felicitous models, such as (8), Prolog usually goes to an infinite loop.

6 Prolog Rules as Defaults

The idea of a self-verifying hypothesis is older than the definition of a felicitous model: Raymond Reiter [31] used it to define the semantics of his nonmonotonic logic for default reasoning.

To specify a *default theory* in the sense of Reiter, we choose

- a set of first-order sentences (similar to axioms in a first-order theory), and
- a set of *defaults*—expressions of the form

$$\frac{F \ : \ \mathsf{M}\, G_1, \ldots, \mathsf{M}\, G_m}{H}, \tag{11}$$

where $F, G_1, \ldots, G_m, H$ are first-order formulas.

Reiter explains the intuition behind the notation for defaults by saying that M is to be read as "it is consistent to assume", and that default (11) says: if F is believed, and if each of $G_1, \ldots G_m$ can be consistently believed, then H is believed.

After giving an informal explanation, Reiter says in the introduction to his paper, a question remains:

> exactly what is meant by the consistency requirement associated with a default? Consistent with what? Providing an appropriate formal definition of this consistency requirement is perhaps the thorniest issue in defining a logic for default reasoning ... For the time being a good intuitive interpretation is to view this consistency requirement with respect to all of the first order facts about the world, together with all of the other beliefs sanctioned by all of the other default rules in force.

The definition of an *extension* for a default theory, given in the main part of the paper, says essentially than an extension is a self-verifying set of beliefs, to use Fine's expression.

Nicole Bidoit and Christine Froidevaux [4] proposed to represent a rule of form (4) by the default (11) in which

– F is the conjunction of the positive literals in the antecedent of (4),
– $G_1, \dots, G_m$ are all negative literals in the antecedent of (4),
– H is the consequent of (4).

For example, this transformation turns the last rule of program (5) into the default

$$\frac{p_2 \ : \ \mathsf{M}\, \neg p_3}{p_4}.$$

For any set Π of rules of form (4), the extensions for the default theory obtained in this way from Π correspond to the felicitous models of Π. In this sense, the reduction of logic programs to default theories invented by Bidoit and Froidevaix justifies negation as failure in the same way as the definition of a felicitous model. But the characterization of preferred models based on that reduction is more complicated than Fine's because it is indirect: to understand it, one needs to be familiar with Reiter's default logic.

7 Prolog Rules as Formulas of Autoepistemic Logic

Autoepistemic logic is a nonmonotonic logic invented by Robert Moore [26] for the purpose of

> modeling the beliefs of ideally rational agents who reflect on their own beliefs ... The language of autoepistemic logic is that of ordinary propositional logic, augmented by a modal operator L. We want formulas of the form $\mathsf{L}P$ to receive the intuitive interpretation of "P is believed" or "I believe P." For example, $P \to \mathsf{L}P$ could be interpreted as saying "If P is true, then I believe that P is true."

About sets A and E of formulas in this language Moore says that E as a *stable expansion* of A if E is the set of all consequences (in the sense of classical propositional logic) of

$$A \cup \{\mathsf{L}P : P \in E\} \cup \{\neg \mathsf{L}P : P \notin E\}.$$

This condition can be viewed as saying that stable expansions are self-justifying, to use Fine's expression again.

Moore relates this use of the word "stable" to an unpublished manuscript by Robert Stalnaker, who talked about a state of belief in which "no further conclusions could be drawn by an ideally rational agent."

Michael Gelfond [15] proposed to translate rules of form (4) into the language of autoepistemic logic by replacing every negative literal $\neg A$ with the formula $\neg \mathsf{L}A$. For example, this translation turns the last rule of program (5) into the formula

$$p_2 \wedge \neg \mathsf{L}p_3 \to p_4.$$

Combined with the definition of a stable expansion quoted above, this translation leads to the same choice of preferred models as the definition of a felicitous model.

The results of Gelfond's paper show that for any stratified program his translation gives the same preferred model as the iterated fixpoint construction.

8 Stable Models

Gelfond's translation produces formulas of a very simple syntactic form, and in application to such formulas the definition of a stable expansion can be replaced by a simpler definition, which does not refer to the consequence relation of propositional logic. This is the idea that has led to the definition of a stable model [17].

Stable models are usually characterized in terms of reducts. The *reduct* of a set Π of rules of form (4) with respect to a set M of atoms is obtained from Π by deleting

- every rule that has a negative literal $\neg A$ in the antecedent such that $A \in M$, and
- all negative literals in the antecedents of the remaining rules.

For example, the reduct of program (5) with respect to M is

$$p_1,$$
$$p_1 \to p_2$$

if p_3 belongs to M. If p_3 does not belong to M then the reduct is (9).

The reduct is obviously a set of rules of form (1). If M is the minimal model of the reduct of Π with respect to M then we say that M is a *stable model* of Π.

This is clearly equivalent to Fine's description of felicitous models.

9 Simple Implications

The literature on stable models describes many ways to extend the definition of the reduct to formulas that are syntactically more general than (4). Some of these proposals are motivated by practical needs of answer set programming; others by the desire to look at stable models in a more abstract way and clarify the essential features of this concept. In this section we discuss one of these generalizations [24], which may be of interest from both points of view.

A *simple implication* is a propositional formula of the form $F \to G$, where F and G are formed from atoms and the logical constants $\top$ (true) and $\bot$ (false) using conjunctions, disjunctions, and negations. A subformula of a simple implication is *critical* if it begins with negation and is not in the scope of another negation. For instance, a rule of form (4) is a simple implication, and its critical parts are the negative literals in its antecedent.

The *reduct* of a set Π of simple implications with respect to a set M of atoms is obtained from Π by replacing each critical subformula C of each implication with $\top$ if M satisfies C, and with $\bot$ otherwise. If M is a minimal model of the reduct of Π with respect to M then we say that M is a *stable model* of Π.

In application to sets of formulas of form (4), this is equivalent to the definition of a stable model given in Section 8. If a set Π of simple implications does not contain negation then its reduct with respect to any set M of atoms is Π

itself, so that the word "stable" in this case has the same meaning as "minimal." For instance, the stable models of the set

$$\begin{array}{l} p \vee q, \\ q \to r \end{array} \tag{12}$$

are its minimal models $\{p\}$ and $\{q, r\}$. (We identify $p \vee q$ with the simple implication $\top \to p \vee q$.)

Each of the formulas

$$p \vee \neg p \tag{13}$$

and

$$\neg\neg p \to p \tag{14}$$

has two stable models, $\emptyset$ and $\{p\}$.

Second digression on intuitionistic logic. If two sets of simple implications are intuitionistically equivalent then they have the same stable models. Intuitionistic logic is not, however, the strongest propositional logic with this property; this assertion holds also for the 3-valued logic introduced by Arend Heyting [18] as a technical device for the purpose of demonstrating that intuitionistic logic is weaker than classical. Heyting remarks that the truth values of this superintuitionistic logic "can be interpreted as follows: 0 denotes a correct proposition, 1 denotes a false proposition, and 2 denotes a proposition that cannot be false but whose correctness is not proved." That logic is known by many names; it is often called the logic of here-and-there, because it can be described by Kripke models with two worlds.

Unlike intuitionistic logic, the logic of here-and-there satisfies De Morgan's law: $\neg(p \wedge q)$ is equivalent to $\neg p \vee \neg q$. The law of excluded middle (13) is equivalent to the law of double negation (14) in the logic of here-and-there; intuitionistically, it is equivalent to the conjunction of (14) with the weak law of excluded middle $\neg p \vee \neg\neg p$.

The relationship between stable models, intuitionistic logic, and the logic of here-and-there was studied by David Pearce and his co-authors [30, 23].

10 Answer Set Solvers

Answer set solvers are software systems that calculate stable models of logic programs. The first such system, SMODELS [28], was followed by DLV [20], CLINGO [13], and several others.

When applied to a program consisting of rules of form (4), an answer set solver never goes to an infinite loop. If a program has several stable models then a solver can be expected to find all of them. For instance, program (8) can be written in the input language of a typical answer set solver as

```
p(1) :- not p(2).
p(2) :- not p(1).
```

The output produced in response to this input may look like this:

```
Answer: 1
p(1)
Answer: 2
p(2)
```

When SMODELS was first presented to the computer science community, the reaction was mixed. The fact that program (8) has two stable models seemed irrelevant, because it does not tell us anyting about the functionality of Prolog. Who would need software that calculates stable models of programs like this? The attitude changed when answer set programming was invented; we will talk about uses of answer set solvers in Sections 12 and 13. Twenty years after the publication of the SMODELS paper, the Association for Logic Programming honored its authors with the prestigious test-of-time award.

Most answer set solvers can handle rules that are syntactically more general than (4). The head of a rule in the input language of CLINGO can be a disjunction of atoms and negated atoms; for instance, the formula

$$p_1 \wedge p_2 \wedge \neg p_3 \rightarrow q_1 \vee q_2 \vee \neg q_3$$

can be written in this language as

```
q(1), q(2), not q(3) :- p(1), p(2), not p(3).
```

As this example shows, the comma in this language represents conjunction when it is used in the body of a rule, and disjunction in the head. The empty head represents the consequent $\bot$; for instance, the formula

$$p \wedge \neg q \rightarrow \bot$$

can be written as

```
:- p, not q.
```

Furthermore, an atom in a CLINGO rule can be preceded by two negations; for instance, formula (14) can be written as

```
p :- not not p.
```

Equivalent transformations of the logic of here-and-there allow us to convert any simple implication into a set of CLINGO rules. It follows that CLINGO can be used, in principle, to calculate the stable models of any finite set of simple implications. For example, the simple implication

$$\neg(p_1 \wedge \neg p_2 \wedge \neg\neg p_3) \wedge q \rightarrow r \tag{15}$$

is equivalent in the logic of here-and-there to the set consisting of three implications:

$$\begin{array}{l} \neg p_1 \wedge q \rightarrow r, \\ \neg\neg p_2 \wedge q \rightarrow r, \\ \neg p_3 \wedge q \rightarrow r. \end{array}$$

Consequently formula (15) can be represented in the input language of CLINGO by the rules

```
r :- not p(1), q.
r :- not not p(2), q.
r :- not p(3), q.
```

11 Rules with Variables

Input languages of most answer set solvers allow programs to use variables. In the tradition of Prolog, variables in answer set programming are denoted by capital letters or, more generally, by character strings that begin with a capital letter. A rule containing variables can be understood as a schematic expression that represents all its "ground instances"—the rules obtained from it by substituting constants (such as integers or symbolic constants) for all variables. For example, the last rule of the program

```
p(1).  p(2).
q(X) :- p(X).
```

stands for the set of rules of the form

```
q(c) :- p(c).
```

for all constants `c`. Thus the program above represents an infinite set of rules of form (1), and its minimal model

```
p(1)  p(2)  q(1)  q(2)
```

is the output that will be produced in response to it by an answer set solver.

As another example, consider the CLINGO program

```
p(1).  p(2).
q(X), not q(X) :- p(X).
```

The head of its last rule is syntactically similar to the excluded middle formula (13). This rule says, informally speaking: for every element X of set p, decide arbitrarily whether or not to include X in set q. The program has 4 stable models:

```
Answer: 1
p(1) p(2)
Answer: 2
p(1) p(2) q(2)
Answer: 3
p(1) p(2) q(1)
Answer: 4
p(1) p(2) q(1) q(2)
```

12 Combinatorial Search

Answer set solvers are often used to solve combinatorial search problems. In such a problem, the goal is to find a solution among a large but finite number of possibilites. Looking for a satisfying truth assignment for a given propositional formula is a standard example of combinatorial search. The set of truth assignments is finite but large: its size is exponential in the size of the input. Solving Sudoku is another example of combinatorial search.

The answer set programming approach to combinatorial search is to encode the problem as a logic program so that the stable models of the program correspond to the objects that we want to find. Then we run an answer set solver on this encoding to find a solution.

Consider, for instance, the exact cover problem: given a collection C of subsets of a finite set U, find a subcollection C^* of C such that each element in U is contained in exactly one subset in C^*. If, for example, U is $\{1,\ldots,4\}$, and C consists of the sets

$$P = \{1,2,3\},\ Q = \{2,3,4\},\ R = \{1,3\},\ S = \{2,3\},\ T = \{2,4\},$$

then $\{R,T\}$ is an exact cover. Deciding whether an exact cover exists is one of Richard Karp's twenty-one NP-complete decision problems [19].

The exact cover problem can be encoded in the language of CLINGO by four rules:

```
c_star(S), not c_star(S) :- in(X,S).
covered(X) :- in(X,S), c_star(S).
:- in(X,S), not covered(X).
:- in(X,S1), in(X,S2), c_star(S1), c_star(S2), S1!=S2.
```

This encoding assumes that the collection C is described by a group of atoms formed using the binary predicate symbol `in`:

```
in(1,p). in(2,p). in(3,p). in(2,q). in(3,q). in(4,q).
in(1,r). in(3,r). in(2,s). in(3,s). in(2,t). in(4,t).
```
(16)

It assumes also that every element of U belongs to at least one element of the collection C (otherwise the problem is not solvable).

The four rules above encode the exact cover problem in the sense that exact covers are in a 1-1 correspondence with the stable models of the program obtained by adding these rules to the atoms describing C. For example, the program obtained by adding these rules to atoms (16) has a unique stable model, and that model includes the atoms `c_star(r)`, `c_star(t)`.

The first rule of the program says, informally speaking, that for every set `S` from the collection C, we may decide arbitrarilty whether or not to include it in the set `c_star`. The second rule defines `covered` as the set of all objects `X` that belong to at least one set `S` from `c_star`. Adding the third rule to the emerging program eliminates the stable models in which some object `X` from one of the

members `S` of C is not covered. Finally, the last rule eliminates the stable models in which some object `X` belongs to two distinct members `S1`, `S2` of `c_star`. (The condition `S1!=S2` in the body of the rule indicates that the constants substituted for the variables `S1` and `S2` in the process of forming ground instances of the rule should not be equal.)

13 Answer Set Programming

Theory of knowledge representation is a subfield of artificial intelligence that studies representing declarative knowledge in a form that can be used by computers. Answer set programming is the approach to knowledge representation based on the language of logic programs under the stable model semantics. Encoding the definition of an exact cover by CLINGO rules in Section 12 is an example of this style of knowledge representation, and thus an example of answer set programming. The concept of a stable model helped researchers in the area of knowledge representation solve the frame problem [32, 21] and other difficult problems in artificial intelligence.

The earliest publication on the use of SMODELS for solving an important computational problem is a paper about planning [7]. Numerous applications of answer set programming and answer set solvers are discussed in recent surveys [9, 10].

The input languages of modern answer set solvers include many constructs that facilitate their use for solving practical problems, and defining the semantics of these constructs sometimes involves extending the concept of a stable model beyond the class of simple implications discussed in Section 9. For instance, the description of some syntactic features of the language of CLINGO [12] refers to stable models of propositional formulas with infinite conjunctions and disjunctions, defined by Miroslaw Truszczynski [33].

The mathematics of stable models has been the subject of many publications, and results in this area are used to prove the correctness of encodings. For example, the informal discussion in the last paragraph of Section 12 can be turned into a proof of the fact that the stable models of the program under consideration are in a 1-to-1 correspondence with exact covers. Extensive research has been done also on the methodology of answer set programming and on the design of solvers. Biannual answer set programming competitions are organized to assess the state of the art [14]. Several textbooks on answer set programming are available today [3, 13, 16, 22], and the *AI Magazine* has published a special issue on answer set programming [1].

The theory of stable models is an example of how "pure" research, motivated by the desire to understand, clarify, and justify, can contribute to the creation of "industrial strength" software.

14 Acknowledgements

Thanks to Michael Gelfond for comments on a draft of this paper.

References

1. *AI Magazine*, 37(3) (2016). Special Issue on Answer Set Programming
2. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In: J. Minker (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann, San Mateo, CA (1988)
3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
4. Bidoit, N., Froidevaux, C.: Minimalism subsumes default logic and circumscription in stratified logic programming. In: Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, pp. 89–97 (1987)
5. Brooks, D.R., Erdem, E., Erdoğan, S.T., Minett, J.W., Ringe, D.: Inferring phylogenetic trees using answer set programming. Journal of Automated Reasoning **39**, 471–511 (2007)
6. Clark, K.: Negation as failure. In: H. Gallaire, J. Minker (eds.) Logic and Data Bases, pp. 293–322. Plenum Press, New York (1978)
7. Dimopoulos, Y., Nebel, B., Koehler, J.: Encoding planning problems in nonmonotonic logic programs. In: S. Steel, R. Alami (eds.) Proceedings of European Conference on Planning, pp. 169–181. Springer (1997)
8. van Emden, M., Kowalski, R.: The semantics of predicate logic as a programming language. Journal of ACM **23**(4), 733–742 (1976)
9. Erdem, E., Gelfond, M., Leone, N.: Applications of Answer Set Programming. AI Magazine **37**, 53–68 (2016)
10. Falkner, A., Friedrich, G., Schekotihin, K., Taupe, R., Teppan, E.: Industrial applications of Answer Set Programming. Künstliche Intelligenz **32**, 165–176 (2018)
11. Fine, K.: The justification of negation as failure. In: Proceedings of the Eighth International Congress of Logic, Methodology and Philosophy of Science, pp. 263–301. North Holland (1989)
12. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract Gringo. Theory and Practice of Logic Programming **15**, 449–463 (2015)
13. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers (2012)
14. Gebser, M., Maratea, M., Ricca, F.: The design of the seventh answer set programming competition. In: Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning, pp. 3–9. Springer (2017)
15. Gelfond, M.: On stratified autoepistemic theories. In: Proceedings of National Conference on Artificial Intelligence (AAAI), pp. 207–211 (1987)
16. Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press (2014)
17. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: R. Kowalski, K. Bowen (eds.) Proceedings of International Logic Programming Conference and Symposium, pp. 1070–1080. MIT Press (1988)
18. Heyting, A.: Die formalen Regeln der intuitionistischen Logik. Sitzungsberichte der Preussischen Akademie von Wissenschaften. Physikalisch-mathematische Klasse pp. 42–56 (1930)
19. Karp, R.: Reducibility among combinatorial problems. In: R.E. Miller, J.W. Thatcher (eds.) Complexity of Computer Computations, pp. 85–103. Plenum (1972)

20. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3), 499–562 (2006)
21. Lifschitz, V.: The dramatic true story of the frame default. Journal of Philosophical Logic **44**(2), 163–176 (2015)
22. Lifschitz, V.: Answer Set Programming. Springer (2019)
23. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic **2**, 526–541 (2001)
24. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence **25**, 369–389 (1999)
25. Marek, V., Truszczynski, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer Verlag (1999)
26. Moore, R.: Possible-world semantics for autoepistemic logic. In: Proceedings 1984 Non-monotonic Reasoning Workshop (1984)
27. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**, 241–273 (1999)
28. Niemelä, I., Simons, P.: Efficient implementation of the well-founded and stable model semantics. In: Proceedings Joint International Conference and Symposium on Logic Programming, pp. 289–303 (1996)
29. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-Prolog decision support system for the Space Shuttle. In: Proceedings of International Symposium on Practical Aspects of Declarative Languages (PADL), pp. 169–183 (2001)
30. Pearce, D.: A new logical characterization of stable models and answer sets. In: J. Dix, L. Pereira, T. Przymusinski (eds.) Non-Monotonic Extensions of Logic Programming (Lecture Notes in Artificial Intelligence 1216), pp. 57–70. Springer (1997)
31. Reiter, R.: A logic for default reasoning. Artificial Intelligence **13**, 81–132 (1980)
32. Shanahan, M.: Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia. MIT Press (1997)
33. Truszczynski, M.: Connecting first-order ASP and the logic FO(ID) through reducts. In: E. Erdem, J. Lee, Y. Lierler, D. Pearce (eds.) Correct Reasoning: Essays on Logic-Based AI in Honor of Vladimir Lifschitz, pp. 543–559. Springer (2012)
34. Van Gelder, A.: Negation as failure using tight derivations for general logic programs. In: J. Minker (ed.) Foundations of Deductive Databases and Logic Programming, pp. 149–176. Morgan Kaufmann, San Mateo, CA (1988)