From Felicitous Models to Answer Set Programming

Vladimir Lifschitz

To the memory of Brian Michael Goldberg

1 Introduction

Felicitous models were defined by Kit Fine [1989] 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 [Bidoit and Froidevaux, 1987, Gelfond, 1987, Gelfond and Lifschitz, 1988].[1] The last of these publications 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.

Years later, sophisticated software systems for generating answer sets were designed, and they became the basis of a new programming paradigm, called answer set programming [Marek and Truszczynski, 1999, Niemelä, 1999]. That programming method has been used for solving computational problems in a variety of areas—from spacecraft design [Nogueira *et al.*, 2001] to historical linguistics [Brooks *et al.*, 2007].

Vladimir Lifschitz
Computer Science Department, University of Texas at Austin, Austin, TX 78712, USA. e-mail: vl@cs.utexas.edu

[1] Fine's paper was presented at the Eighth International Congress of Logic, Methodology and Philosophy of Science, held in 1987. Bidoit and Froidevaix described their work at the Second Annual IEEE Symposium on Logic in Computer Science the same year. Gelfond's paper was given at the AAAI National Conference on Artificial Intelligence, in 1987 as well. The paper by Gelfond and Lifschitz was initially submitted to the Seventh ACM Symposium on Principles of Database Systems and became "one of the papers which time considerations prevented from being presented at the Symposium"; in 1988 it was contributed to the International Logic Programming Conference and Symposium.

In this chapter, after a brief introduction to Prolog (Section 2), we trace the early history of felicitous models (Sections 3–10) and then talk about contributions of that idea to computer science (Sections 11–14). This is a piece of intellectual history, and there are no new technical results here. Parts of the chapter are written in an informal style, but in several places we give precise definitions and precise statements of important theorems, or tell the reader where details can be found.

2 A Prolog Program

A Prolog program consists of *rules*—syntactic objects that are closely related to formulas of a first-order language.

As an example, consider a Prolog solution to a simple computational problem [Lifschitz, 2019, Section 1.2]. We are given a table showing the population sizes of several countries, for instance:

Country	France	Germany	Italy	United Kingdom
Population (million)	65	83	61	64

Table 1 Population of European countries in 2015.

The goal is to make the list of all countries inhabited by more people than the United Kingdom. We will call such countries "large."

The list of large countries can be generated by the Prolog program consisting of a single rule:

```
large(C) :- size(C,S1), size(uk,S2), S1 > S2.    (1)
```

This rule has two parts—the *head* `large(C)` and the *body*

```
size(C,S1), size(uk,S2), S1 > S2
```

—separated by the "colon-dash" symbol, which reads "if." The end of a rule is indicated by a period. Character strings that begin with a capital letter (in this case, `C`, `S1`, and `S2`) are variables. Since `uk` is the name of a specific object and not a variable, it is not capitalized. The symbol `size` in the body represents the binary relation that holds between a country and its population size. Thus rule (1) can be translated into English as follows:

A country C is large
if the population size of C is S_1, the population size of the UK is S_2, and $S_1 > S_2$.

To generate the list of large countries using rule (1), we encode the input—Table 1—as a collection of additional rules:

```
size(france,65).   size(germany,83).
size(italy,61).    size(uk,64).
```
(2)

Each of these additional rules is "a head without a body."

A Prolog system will load a file consisting of rules (1) and (2), in any order, and display the prompt `?-` that invites the user to submit "queries"—questions that can be answered on the basis of the given information. The query `large(C)` would be understood as the request to find a value of `C` that has the property `large`, and the system would respond:

```
C = france.
```

If the user requests another value of `C` with this property, the answer will be

```
C = germany.
```

To a request for a third solution the system will reply `no` (no more large countries).

The first Prolog system was developed in 1972 by Alain Colmerauer and Phillipe Roussel at Aix-Marseille University. It was used originally for natural language processing. Its name is an abbreviation for *programmation en logique* (*programming in logic*).

3 Minimal Models

Propositional formulas of the form

$$A_1 \wedge \cdots \wedge A_n \to A_{n+1} \qquad (n \geq 0), \tag{3}$$

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

$$\begin{array}{l} p_1, \\ p_1 \to p_2, \\ p_2 \wedge p_3 \to p_4 \end{array} \tag{4}$$

can be represented by the Prolog program

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

A Prolog system will tell us that p_1 and p_2 are the only atoms entailed by formulas (4).

The answers given by Prolog in examples like this can be interpreted also in terms of minimal models [van Emden and Kowalski, 1976]. In the theory of Prolog 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 (4), 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 (4) as well. The model $\{p_1, p_2\}$ differs from the others in that it is minimal with respect to set inclusion: its proper subsets are not models of (4).

We can say that in application to a set of formulas of form (3) a Prolog system calculates the minimal model of that set. To be precise, it 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

$$\begin{array}{l} p_1 \to p_2, \\ p_2 \to p_1. \end{array} \tag{5}$$

The minimal model of a set Π of formulas of form (3) 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 (4); 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 (4); this is the minimal model.

Formulas (5) are satisfied by the empty set, so that the process of accumulating atoms stops in this case at the very beginning.

4 Negation as Failure

The observations in Section 3 apply to Prolog rules that correspond to formulas of form (3). How can we extend them to propositional formulas of the form

$$L_1 \wedge \cdots \wedge L_n \to H \qquad (n \geq 0), \tag{6}$$

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 discovery (or should we say "invention"?) of felicitous models.

Consider, for instance, the formulas

$$\begin{array}{l} p_1, \\ p_1 \to p_2, \\ p_2 \wedge \neg p_3 \to p_4 \end{array} \tag{7}$$

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 (7) 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 a Prolog system 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 (7) 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 (7) 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 (7) 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 (7).

Explanations in terms of failure of an attempt to use rules comports with the way Prolog programmers think; they say that the symbol $\backslash+$ represents "negation as failure" [Clark, 1978]. But how can we turn such an informal explanation into a mathematical definition?

Digression on intuitionistic logic. If we replace the last of formulas (7) by the equivalent formula

$$p_2 \wedge \neg p_4 \rightarrow p_3$$

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 10 we will say more about the relationship between preferred models and intuitionistic logic.

5 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 programs that are "stratified," or "free from recursive negation" [Apt *et al.*, 1988, Van Gelder, 1988].

We will sometimes identify a Prolog rule with the corresponding propositional formula. To stratify a finite set Π of rules of form (6) means to partition it into subsets $\Pi_1, \ldots, \Pi_k$ so that if a rule (6) belongs to Π_i $(1 \leq i \leq k)$ then

(a) whenever a literal L_j in the antecedent of (6) 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 (6) 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 (3) is stratified: include the entire set in one stratum Π_1. Program (7) is stratified using a single stratum as well. The program consisting of the rules

$$\neg p_i \to p_{i+1} \qquad (1 \leq i \leq k), \tag{8}$$

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

$$\neg p \to p \tag{9}$$

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

$$\begin{array}{l} \neg p_1 \to p_2, \\ \neg p_2 \to p_1. \end{array} \tag{10}$$

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 3. 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 (7), then it is never included.

For example, constructing the preferred model of program (8) 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 (8) 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 [Apt *et al.*, 1988, Theorem 11].

6 Felicitous Models

To characterize preferred models in the general case, "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 [Fine, 1989, page 285].

Take, for example, program (7), and consider the hypothesis that p_3 is false. Then the conjunctive term $\neg p_3$ in the last line of (7) is true, and "detaching" it turns (7) into the program

$$\begin{array}{l} p_1, \\ p_1 \to p_2, \\ p_2 \to p_4. \end{array} \tag{11}$$

This is a set of rules of form (3), and its minimal model is

$$\{p_1, p_2, p_4\}. \tag{12}$$

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 (12) 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 [Fine, 1989, 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 (7) and consider now the hypothesis that p_3 is not among the false atoms. Then the antecedent in last line of (7) 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 (7), set (12) is the only felicitous model—the only set of atoms that leads to neither gaps not gluts. Rule (9) has no felicitous models, and program (10) has two: $\{p_1\}$ and $\{p_2\}$.

The concept of a felicitous model describes the behavior of Prolog systems in the following sense: *if Π is a program consisting of rules of form (6) that has a unique stable model, and the system 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 (9), or several felicitous models, such as (10), Prolog usually goes to an infinite loop.

Fine's exposition is somewhat informal—he was interested in motivating his idea, explaining the intuition behind it, and comparing it with other approaches to the problem, rather than in converting it into a concise definition of the kind found in mathematically oriented publications. But in Section 6, which begins with the words "I shall now present my own proposal," mathematical precision lurks behind the veil of informality.

Fine's proposal can be defined more formally using the following auxiliary definition. The *reduct* of a set Π of rules of form (6) 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 (7) 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 (11).

The reduct is obviously a set of rules of form (3). If M is the minimal model of the reduct of Π with respect to M then, according to Fine, M is a *felicitous model* of Π.

7 Prolog Rules as Defaults

The idea of a self-verifying hypothesis is older than the definition of a felicitous model: Raymond Reiter [1980] 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

$$\frac{F \ : \ \mathsf{M}G_1, \ldots, \mathsf{M}G_m}{H}, \tag{13}$$

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 (13) 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 [Reiter, 1980, Sections 2.2 and 7.1], says essentially than an extension is a self-verifying set of beliefs, to use Fine's expression.

Nicole Bidoit and Christine Froidevaux [1987] proposed to represent a rule of form (6) by the default (13) in which

- F is the conjunction of the positive literals in the antecedent of (6),
- $G_1, \ldots, G_m$ are all negative literals in the antecedent of (6),
- H is the consequent of (6).

For example, this transformation turns the last rule of program (7) into the default

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

For any set Π of rules of form (6), 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 Froidevaux 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.

8 Prolog Rules as Formulas of Autoepistemic Logic

Autoepistemic logic is a nonmonotonic logic invented by Robert Moore [1984] 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 [1987] proposed to translate rules of form (6) 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 (7) 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.

9 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 [Gelfond and Lifschitz, 1988].

Stable models are usually characterized in terms of reducts, as at the end of Section 6 above. In fact, the description of the reduct there is a quote, almost verbatim, from the stable model paper.

10 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 (6). 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 [Lifschitz *et al.*, 1999], 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 (6) 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 (6), this is equivalent to the definition of a stable model given in Section 9. 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

$$\begin{array}{l} p \vee q, \\ q \to r \end{array} \tag{14}$$

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

$$p \vee \neg p \tag{15}$$

and

$$\neg\neg p \to p \tag{16}$$

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 [Lifschitz *et al.*, 2001]. 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 [1930] 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 (15) is equivalent to the law of double negation (16) in the logic of here-and-there; intuitionistically, it is equivalent to the conjunction of (16) with the weak law of excluded middle $\neg p \vee \neg\neg p$.

The relationship between stable models and the logic of here-and-there was discovered by David Pearce [1997].

11 Answer Set Solvers

Answer set solvers are software systems that calculate stable models of logic programs. The first such system, SMODELS [Niemelä and Simons, 1996], was followed by DLV [Leone *et al.*, 2006], CLINGO [Gebser *et al.*, 2012], and several others.

When applied to a program consisting of rules of form (6), 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 (10) 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 (10) has two stable models seemed irrelevant, because it does not tell us anything about the functionality of Prolog systems. 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 13 and 14. 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 (6). The head of a rule in the input language of CLINGO can be a disjunction of atoms and negated atoms; for instance, the formula

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

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

$$p \wedge \neg q \to \bot$$

can be written as

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

Furthermore, an atom in a CLINGO rule can be preceded by two negations; for instance, formula (16) 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

$$\neg(p_1 \wedge \neg p_2 \wedge \neg\neg p_3) \wedge q \to r \tag{17}$$

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

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

Consequently formula (17) 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.
```

12 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 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 (3), 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 (15). 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)
```

13 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 puzzles is another example of combinatorial search.

The answer set programming approach to combinatorial search is to encode the problem as a set of rules so that its stable models 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 of U is contained in exactly one subset in C^*. If, for example, U is $\{1,\dots,4\}$, 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 the examples of difficult ("NP-complete") decision problems from the classical paper by Richard Karp [1972].

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

```
1 c_star(S), not c_star(S) :- in(X,S).
2 covered(X) :- in(X,S), c_star(S).
3 :- in(X,S), not covered(X).
4 :- 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(q.3). in(q,4).
in(1,r). in(3,r). in(2,s). in(3,s). in(2,t). in(4,t).
```
(18)

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 in the listing 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 (18) has a unique stable model, and that model includes the atoms `c_star(r)`, `c_star(t)`.

The rule in Line 1 of the listing 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`. Line 2 defines `covered` as the set of all objects `X` that belong to at least one set `S` from `c_star`. Adding Line 3 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, adding Line 4 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.)

14 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 stable model semantics. Encoding the definition of an exact cover by CLINGO rules in Section 13 is an example of this style of knowledge representation, and thus an example of answer set programming.

The earliest publication on the use of SMODELS for solving an important computational problem is a paper about plan generation [Dimopoulos *et al.*, 1997]. Numerous applications of answer set programming and answer set solvers are discussed in recent surveys [Erdem *et al.*, 2016, Falkner *et al.*, 2018].

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 10. For instance, the description of some syntactic features of the language of CLINGO [Gebser *et al.*, 2015] refers to stable model models of propositional formulas with infinite conjunctions and disjunctions, defined by Miroslaw Truszczynski [2012].

The mathematics of stable models has been the subject of many publications, and results in this area can be used to prove the correctness of encodings. For example, the informal discussion in the last paragraph of Section 13 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. In some cases, such proofs can be verified using an automated proof assistant [Fandinno *et al.*, 2020]. 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 [Gebser *et al.*, 2017].

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.

15 Acknowledgements

The author is grateful to Federico Faroldi, Kit Fine, Michael Gelfond, Frederik Van De Putte, and the anonymous referees for their comments on drafts of this chapter.

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 Pro-*

gramming, pages 89–148. Morgan Kaufmann, San Mateo, CA, 1988.

[Bidoit and Froidevaux, 1987] Nicole Bidoit and Christine Froidevaux. Minimalism subsumes default logic and circumscription in stratified logic programming. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, pages 89–97, 1987.

[Brooks *et al.*, 2007] Daniel R. Brooks, Esra Erdem, Selim T. Erdoğan, James W. Minett, and Donald Ringe. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning*, 39:471–511, 2007.

[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.

[Dimopoulos *et al.*, 1997] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in non-monotonic logic programs. In Sam Steel and Rachid Alami, editors, *Proceedings of European Conference on Planning*, pages 169–181. Springer, 1997.

[Erdem *et al.*, 2016] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of Answer Set Programming. *AI Magazine*, 37:53–68, 2016.

[Falkner *et al.*, 2018] Andreas Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich Teppan. Industrial applications of Answer Set Programming. *Künstliche Intelligenz*, 32:165–176, 2018.

[Fandinno *et al.*, 2020] Jorge Fandinno, Vladimir Lifschitz, Patrick Lühne, and Torsten Schaub. Verifying tight programs with Anthem and Vampire. *Theory and Practice of Logic Programming*, 20, 2020.

[Fine, 1989] Kit Fine. The justification of negation as failure. In *Proceedings of the Eighth International Congress of Logic, Methodology and Philosophy of Science*, pages 263–301. North Holland, 1989.

[Gebser *et al.*, 2012] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[Gebser *et al.*, 2015] Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz, and Torsten Schaub. Abstract Gringo. *Theory and Practice of Logic Programming*, 15:449–463, 2015.

[Gebser *et al.*, 2017] Martin Gebser, Marco Maratea, and Francesco Ricca. The design of the seventh answer set programming competition. In *Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 3–9. Springer, 2017.

[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. MIT Press, 1988.

[Gelfond, 1987] Michael Gelfond. On stratified autoepistemic theories. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 207–211, 1987.

[Heyting, 1930] Arend Heyting. Die formalen Regeln der intuitionistischen Logik. *Sitzungsberichte der Preussischen Akademie von Wissenschaften. Physikalisch-mathematische Klasse*, pages 42–56, 1930.

[Karp, 1972] Richard Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, 1972.

[Leone *et al.*, 2006] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[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, 2019] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019.

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

[Moore, 1984] Robert Moore. Possible-world semantics for autoepistemic logic. In *Proceedings 1984 Non-monotonic Reasoning Workshop*, 1984.

[Niemelä and Simons, 1996] Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings Joint International Conference and Symposium on Logic Programming*, pages 289–303, 1996.

[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.

[Nogueira *et al.*, 2001] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the Space Shuttle. In *Proceedings of International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 169–183, 2001.

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

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

[Truszczynski, 2012] Miroslaw Truszczynski. Connecting first-order ASP and the logic FO(ID) through reducts. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning: Essays on Logic-Based AI in Honor of Vladimir Lifschitz*, pages 543–559. Springer, 2012.

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

[Van Gelder, 1988] Allen Van Gelder. Negation as failure using tight derivations for general logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann, San Mateo, CA, 1988.