CS 395T, Spring 2005
Answer Set Programming

Answer Sets

1 Traditional Programs and their Answer Sets

The definition of an answer set was originally proposed as a semantics for Prolog programs with negation, and later extended to more general logic programs. We begin by considering the case of "traditional" programs, which are essentially Prolog programs without variables.

1.1 Syntax

A *traditional rule* is an expression of the form

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

where $n \geq m \geq 0$ and $A_0, \dots, A_n$ are propositional atoms. The atom A_0 is called the *head* of the rule, and the list $A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n$ is its *body*. If the body is empty ($n = 0$) then the rule is called a *fact* and identified with its head A_0.

A *traditional program* is a finite set of traditional rules. For instance,

$$\begin{array}{l} p \\ r \leftarrow p, q \end{array} \tag{2}$$

and

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

are traditional programs.

A traditional rule (1) is *positive* if $m = n$, that is to say, if it has the form

$$A_0 \leftarrow A_1, \dots, A_m. \tag{4}$$

A traditional program is *positive* if each of its rules is positive. For instance, program (2) is positive, and (3) is not.

1.2 The Answer Set of a Positive Program

We will first define the concept of an answer set for positive traditional programs. We need an auxiliary definition: a set X of atoms *satisfies* a positive traditional program Π if, for all rules (4) in Π, $A_0 \in X$ whenever $A_1, \ldots, A_m \in X$. For instance, any positive traditional program is satisfied by the set of the heads A_0 of all its rules (4).

Problem 1.1 For any positive traditional program Π, the intersection of all sets satisfying Π satisfies Π also.

This assertion 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* of Π.

For instance, the sets of atoms satisfying program (2) are

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

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

Intuitively, we can think of (4) as a rule for generating atoms: once you have generated $A_1, \ldots, A_m$, you are allowed to generate A_0. The answer set is the set of all atoms that can be generated by applying rules of the program in any order. For instance, the first rule of (3) allows us to include p in the answer set. The second rule says that we can add r to the answer set if we have already included p and q. Given these two rules only, we can generate no atoms besides p. If we extend program (3) by adding the rule $q \leftarrow p$ then the answer set will become $\{p,q,r\}$.

1.3 Answer Sets of a Program with Negation

To extend the definition of an answer set to arbitrary traditional programs, we will introduce one more auxiliary definition. The *reduct* Π^X of a traditional program Π relative to a set X of atoms is the set of rules (4) for all rules (1) in Π such that $A_{m+1}, \ldots, A_n \notin X$. Thus Π^X is a positive traditional program.

We say that X is an *answer set* of Π if X is the answer set of Π^X (that is, the smallest set of atoms satisfying Π^X).

This definition was proposed in the paper [Gelfond and Lifschitz, 1988], where answer sets were called "stable models." Its idea came from early work on nonmonotonic reasoning [McDermott and Doyle, 1980] and [Reiter, 1980], which was related to logic programming in [Gelfond, 1987].

If Π is positive then, for any X, $\Pi^X = \Pi$. It follows that the new definition of an answer set is a generalization of the definition from Section 1.2:

for any positive traditional program Π, X is the smallest set of atoms satisfying Π^X iff X is the smallest set of atoms satisfying Π.

Example: if Π is (3) and $X = \{q\}$ then the reduct Π^X consists of the single fact q. The answer set of Π^X is $\{q\}$. Consequently, $\{q\}$ is an answer set of Π.

Intuitively, rule (1) allows us to generate A_0 as soon as we generated the atoms $A_1, \dots, A_m$ *provided that none of the atoms* $A_{m+1}, \dots, A_n$ *can be generated using the rules of the program.* There is a vicious circle in this sentence: to decide whether a rule of Π can be used to generate a new atom, we need to know which atoms can be generated using the rules of Π. The definition of an answer set overcomes this difficulty by employing a "fixpoint construction." Take a set X that you suspect may be exactly the set of atoms that can be generated using the rules of Π. Under this assumption, Π has the same meaning as the positive program Π^X. Consider the answer set of Π^X, as defined in Section 1.2. If this set is exactly identical to the set X that you started with then X was a "good guess"; it is indeed an answer set of Π.

Problem 1.2 Check that $\{p\}$ is not an answer set of 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 $\{p, q, r\}$. This question can be also answered using the following general properties of answer sets of traditional programs.

Problem 1.3 If X is an answer set of a traditional program Π then every element of X is the head of one of the rules of Π.

Problem 1.4 (a) For any traditional program Π and any sets X, Y of atoms, if $X \subseteq Y$ then $\Pi^Y \subseteq \Pi^X$. (b) If X is an answer set for a traditional program Π then no proper subset of X can be an answer set of Π.

In application to program (3), the assertion of Problem 1.3 tells us that its answer sets do not contain r, so that we only need to check $\emptyset$ and $\{p, q\}$. By the assertion of Problem 1.4(b), $\emptyset$ cannot be an answer set because it is a proper subset of the answer set $\{q\}$, and $\{p, q\}$ cannot be an answer set because the answer set $\{q\}$ is its proper subset. Consequently, $\{q\}$ is the only answer set of (3).

Problem 1.5 Find an answer set of the program Π_n consisting of n rules

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

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

$$\begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \end{array} \tag{5}$$

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

$$r \leftarrow not\ r \tag{6}$$

has no answer sets.

Problem 1.6 Find all answer sets of the following program, which extends (5) by two additional rules:

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

Problem 1.7 Find all answer sets of the following combination of programs (5) and (6):

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

1.4 Prolog vs. Answer Set Programming

The definition of an answer set for traditional programs can be viewed as a possible definition of a "correct" answer to a query in Prolog.

Let Π be a Prolog program without variables (or the set of ground rules obtained from a Prolog program with variables by replacing each rule by all its ground instances). If Π is a traditional program in the sense of Section 1.1, and it 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 (5), is "bad"—it does not provide an unambiguous specification for the behavior of a Prolog system. Programs without answer sets, such as (6), 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 of 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.

The concept of an answer set is only one of several definitions of the semantics of Prolog with negation proposed 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].

1.5 LPARSE and SMODELS

The most widely used among the answer set solvers available today is SMODELS, with the preprocesor LPARSE. It has been designed and implemented at the Helsinki University of Technology, and it can be downloaded from its web site, `http://www.tcs.hut.fi/Software/smodels/` .

On the machines of the Department of Computer Sciences of the University of Texas at Austin, LPARSE and SMODELS are installed in directories

`/projects/tag/smodels/sun/`

and

`/projects/tag/smodels/linux/` .

If you are using these machines then you may find it convenient to define appropriate aliases, for instance:

```
alias lparse '/projects/tag/smodels/sun/lparse'
alias smodels '/projects/tag/smodels/sun/smodels'
```

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 of the program from Problem 1.6, we create the file

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

called, say, `p1.6` . Then we invoke LPARSE and SMODELS as follows:

```
% lparse p1.6 | 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").

A group of rules that follow a pattern can be often described concisely in the input language of LPARSE using schematic variables. As in Prolog, variables must be capitalized. Consider, for instance, the programs Π_n from Problem 1.5. To describe Π_7, we don't have to write out each of its 7 rules. Instead, let's agree to use, for instance, the symbol `index` to represent numbers between 1 and 7. We can write Π_7 as

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

The first two lines declare `I` to be a variable ranging over $\{1, \ldots, 7\}$. The auxiliary symbols used to describe the ranges of variables, such as `index`, are called domain predicates. Grounding—translating schematic expressions, such as `p(I) :- not p(I+1)`, into sets of rules—is the main computational task performed by LPARSE.

Instead of declaring a variable, we can specify its range in the body of the rule in which it is used:

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

The family of programs Π_n $(n = 1, 2, \ldots)$ can be described by a program schema with the parameter n:

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

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

```
% lparse -c n=7 p1.5 | smodels 0
```

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. Information on the extents of domain predicates in the output can be suppressed using the LPARSE option `-d none`:

```
% lparse -c n=7 -d none p1.5 | smodels 0
```

Problem 1.8 Use SMODELS to verify that Π_n has a unique answer set for $n = 10$ and $n = 100$.

Problem 1.9 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.

In Section 3 we will discuss several other useful features of the input language of LPARSE. A complete manual can be downloaded from the SMODELS web site.

2 General Definition of an Answer Set

In a series of papers on answer sets, the definition of that concept for traditional programs reproduced in Section 1 was extended to more general programs. These generalizations play an important role is answer set programming. In [Gelfond and Lifschitz, 1991], the head of a rule is allowed to be a disjunction of several atoms, rather than a single atom, for instance:

$$p; q \leftarrow r \tag{7}$$

(in logic programming, disjunction is usually written as a semicolon, just as conjunction is written as a comma).[1] In particular, the disjunction in the head can be empty:

$$\leftarrow r. \tag{8}$$

[1] Another extension of the syntax of logic programs proposed in that paper is the use of a second kind of negation. It is called "classical," "strong," or "true" negation, to distinguish it from "negation as failure" *not*, and is sometimes denoted by $\sim$.

In a further generalization [Lifschitz and Woo, 1992], the head of a rule may contain negated atoms:

$$p; \mathit{not}\ q \leftarrow r. \tag{9}$$

According to the still more general "nested" syntax of rules [Lifschitz *et al.*, 1999], the head and the body of a rule are arbitrary formulas built from atoms using any number of conjunctions, disjunctions and negations in any order:

$$(p; \mathit{not}\ q), r \leftarrow \mathit{not}(q, \mathit{not}\ r). \tag{10}$$

Here rules become essentially arbitrary implications that contain no other implications in their antecedents (bodies) and consequents (heads).

Furthermore, the analysis of answer sets in [Pearce, 1997] suggests a simple generalization of the definition of the reduct to arbitrary propositional formulas, including even those that contain nested implications [Ferraris, 2005]. The theory of answer sets presented below follows the approach of the last paper, except that we limit attention to finite programs.

2.1 Rules and Programs as Formulas

The syntax and semantics of propositional formulas are reviewed in the appendix (Section A.1). We use

$$\begin{array}{rll} F, G & \text{as alternative notation for} & F \wedge G, \\ F; G & \text{as alternative notation for} & F \vee G, \\ F \leftarrow G & \text{as alternative notation for} & G \rightarrow F, \\ \mathit{not}\ F \text{ and } \leftarrow F & \text{as alternative notation for} & \neg F. \end{array}$$

Given these conventions, rules of various kinds mentioned above can be viewed as formulas. For instance, a traditional rule (1) with $n > 0$ is alternative notation for the implication

$$(A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n) \rightarrow A_0.$$

We can say that a traditional rule is an implication whose antecedent is a conjunction of literals, and the consequent is an atom. Rules (7)–(10) are identical to the formulas

$$\begin{gathered} r \rightarrow (p \vee q), \\ \neg r, \\ r \rightarrow (p \vee \neg q), \\ \neg(q \wedge \neg r) \rightarrow ((p \vee \neg q) \wedge r). \end{gathered}$$

On the other hand, the formula

$$(p \to q) \to r \tag{11}$$

can be written as a "rule with an embedded implication"

$$r \leftarrow (p \to q)$$

or, alternatively,

$$r \leftarrow (q \leftarrow p).$$

From now on, the terms "formula" and "rule" will be used interchangeably. A *program* is a finite set of rules. We will identify a program with the conjunction of its rules. For instance, traditional program (2) can be written as the formula

$$p \wedge ((p \wedge q) \to r). \tag{12}$$

Note that the definition of satisfaction for positive traditional programs (Section 1.2) is a special case of the usual definition of satisfaction for propositional formulas (Section A.1).

2.2 Reducts and Answer Sets

The *reduct* F^X of a formula F relative to a set X of atoms is the formula obtained from F by replacing each maximal subformula that is not satisfied by X with $\perp$. We say that X is an *answer set* of F if X is a minimal set satisfying F^X.

The mininality of X is understood here in the sense of set inclusion. We can verify that X is an answer set of F as follows:

(i) mark in F the maximal subformulas that are not satisfied by X;

(ii) replace each of these subformulas with $\perp$ (after that, equivalent transformations can be used to simplify the result);

(iii) check that the resulting formula is satisfied by X;

(iv) check that it is not satisfied by any proper subset of X.

For instance, to check that $\{p\}$ is an answer set of (12), we do the following:

(i) mark the maximal subformulas of (12) that are not satisfied by $\{p\}$:

$$p \wedge ((\underline{p \wedge q}) \to \underline{r});$$

(ii) replace these subformulas with $\bot$:

$$p \wedge (\bot \rightarrow \bot);$$

simplify:

$$p;$$

(iii) check that the last formula is satisfied by $\{p\}$;

(iv) check that it is not satisfied by $\emptyset$.

Problem 2.1 For each subset of $\{p, q, r\}$ other than $\{p\}$ check that it is not an answer set of (12).

In application to formulas corresponding to traditional programs, the new definition of the reduct, is, generally, *not* equivalent to the definition from Section 1.3. For instance, the reduct of traditional program (2) relative to any set X of atoms is (12), because that program doesn't contain negation; but the reducts of the corresponding formula (12) are, for the most part, different from (2). However, the new definition of an answer set, when applied to a traditional program written as a formula, turns out to be exactly equivalent to the old definition [Ferraris, 2005]:

Theorem on Traditional Programs. *The answer sets of a traditional program in the sense of Section 1.3 are identical to the answer sets of the corresponding formula.*

In particular, a formula corresponding to a positive traditional program, such as (12), has a unique answer set, which is the smallest set of atoms satisfying that formula.

Problem 2.2 Verify the assertion of the theorem on traditional programs for the one-rule program

$$p \leftarrow not\ q.$$

(Recall that $\neg q$, according to Section A.1, is shorthand for $q \rightarrow \bot$.)

It is easy to see that for any X

$$\bot^X = \bot;$$

$$A^X = \begin{cases} A, & \text{if } X \models A, \\ \bot, & \text{otherwise} \end{cases}$$

(A is an atom);

$$(F \odot G)^X = \begin{cases} F^X \odot G^X, & \text{if } X \models F \odot G, \\ \bot, & \text{otherwise} \end{cases}$$

($\odot$ is $\wedge$, $\vee$ or $\rightarrow$).

These equalities provide an alternative, recursive definition of the reduct.

Problem 2.3 For any formula F and any set X of atoms,

$$(\neg F)^X = \begin{cases} \bot, & \text{if } X \models F, \\ \top, & \text{otherwise.} \end{cases}$$

This fact facilitates computing the reducts of formulas written in terms of negation, as in Problem 2.2. It shows also that, in spite of all the differences between the definition of the reduct for traditional programs and the new definition of the reduct for arbitrary formulas, there is a certain similarity between them. The reduct of a traditional program, as defined in Section 1.3, is a positive program: in the process of constructing the reduct of Π all negative terms in the bodies of the rules of Π are eliminated. Similarly, in the recursive process of constructing the reduct of a formula, all its subformulas of the form $\neg F$ are eliminated: each of them turns either into $\bot$ or into $\top$.

2.3 Answer Set Semantics

It is easy to see that any answer set of F satisfies F. Indeed, if X doesn't satisfy F then the reduct F^X is $\bot$, so that it cannot be satisfied by X. In this sense, the answer set semantics of propositional formulas is stronger than the classical semantics, which is given by the concept of satisfaction. The sets satisfying a formula are sometimes called its "models," and the answer sets of a formula are called its "stable" models.

The classical semantics of propositional formulas is monotonic, in the sense that conjoining a formula with another formula can only make the set of its models smaller: if X satisfies $F \wedge G$ then X satisfies F. The answer set semantics is nonmonotonic: an answer set of $F \wedge G$ does not have to be an answer set of F. This phenomenon is observed even when F and G are positive traditional programs, and it is related to the minimality condition in the definition of an answer set. For instance, the answer set $\{p, q\}$ of $p \wedge q$ is not an answer set of p. The usefulness of nonmonotonicity as a property of knowledge representation formalisms and its relation to minimality conditions were established in early work on circumscription [McCarthy, 1980].

The classical semantics of propositional formulas and the answer set semantics occupy different places in the polynomial hierarchy: satisfiability is an NP-complete property, and the existence of an answer set is Σ_2^P-complete [Eiter and Gottlob, 1993]. This jump in complexity occurs as soon as the concept of an answer set is extended to disjunctive rules, such as (7); for traditional programs (Section 1), the existence of an answer set is in class NP.

Problem 2.4 (a) Formula $p \vee q$ has two answer sets: $\{p\}$ and $\{q\}$. (b) Formula

$$(p \vee q) \wedge (p \leftrightarrow q)$$

has one answer set: $\{p, q\}$.

These facts provide one more example of nonmonotonicity: after appending the conjunctive term $p \leftrightarrow q$, the formula $p \vee q$ gets a new answer set. They show also that disjunction, under the answer set semantics, is sometimes "exclusive," and sometimes not. On the one hand, among the three sets

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

that satisfy $p \vee q$, the third is not an answer set. On the other hand, appending the $p \leftrightarrow q$ to the disjunction makes $\{p, q\}$ an answer set; the disjunction "becomes inclusive."

2.4 Answer Sets and Intuitionistic Equivalence

Problem 2.5 (a) Formula $\neg\neg p$ has no answer sets. (b) Each of the two formulas

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

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

Even though the formulas p and $\neg\neg p$ are equivalent to each other, one of them has an answer set, and the other doesn't. Even though the formulas (13) are equivalent to $\top$, each of them has two answer sets, and $\top$ has only one (empty).

The results of [Pearce, 1997] show, however, that the answer sets of a formula are preserved by *intuitionistically* equivalent transformations (see Section A.3 for the definition):

Theorem on Intuitionistic Equivalence. *If F is intuitionistically equivalent to G then F and G have the same answer sets.*

Problem 2.6 The answer sets of a program are not affected by replacing (a) a rule of the form

$$F, G \leftarrow H$$

with the pair of rules

$$F \leftarrow H$$
$$G \leftarrow H \ ;$$

(b) a rule of the form

$$F \leftarrow G; H$$

with the pair of rules

$$F \leftarrow G$$
$$F \leftarrow H \ .$$

Problem 2.7 The answer sets of a program are not affected by replacing (a) a rule of the form

$$not\ F \leftarrow G$$

with

$$\leftarrow F, G \ ;$$

(b) a rule of the form

$$\leftarrow not\ not\ F$$

with

$$\leftarrow F \ .$$

2.5 Head Atoms

In Section 1.3 we discussed two useful properties of answer sets of traditional programs. First, every element of an answer set of a traditional program is the head of one of its rules (Problem 1.3). Second, according to Problem 1.4(b), the answer sets of a traditional program form an "anti-chain": one of them cannot be a proper subset of another.

The examples from Problem 2.5(b) show that the anti-chain property does not extend to arbitrary formulas. As to the assertion of Problem 1.3, it can be extended to formulas as follows.

Recall that an occurrence of an atom in a formula F is called positive if the number of implications containing that occurrence in the antecedent is even (Section A.1). An occurrence of an atom in F is *strictly positive* if that number is 0, that is to say, if that occurrence does not belong to the antecedent of any implication in F. Clearly, a formula of the form $\neg F$ has no strictly positive occurrences of atoms (recall that $\neg F$ stands for $F \rightarrow \bot$).

We say that A is a *head atom* of F if at least one occurrence of A in F is strictly positive. For instance, the head atoms of a traditional program are the heads of its rules. The only head atom of (9) is p.

Theorem on Head Atoms. *If X is an answer set of a formula F then every element of X is a head atom of F.*

This theorem is due to Paolo Ferraris (personal communication).

Problem 2.8 What does the theorem on head atoms say about the answer sets of formula (11)? Find its answer sets.

3 Choice Rules and Constraints

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 of a logic program. This is often achieved by combining rules of two kinds, studied in this section. A "choice rule" is a program with many answer sets that provides an approximation from above for the collection of sets that we want to describe. The redundant answer sets can be removed by adding "constraints" to this program.

3.1 Choice Formulas

For any finite set Z of atoms, by Z^c we denote the formula

$$\bigwedge_{A \in Z} (A \vee \neg A)$$

(Section A.3).

Problem 3.1 A set X is an answer set of Z^c iff X is a subset of Z.

Thus if Z consists of n atoms then Z^c has 2^n answer sets. Under the answer set semantics, Z^c says: choose for every element of Z arbitrarily whether to include it in the answer set. We will call formulas of the form Z^c *choice formulas*. (The superscript ^{c} is used in this notation because it is the first letter of the word "choice.")

Although LPARSE does not allow us to use conjunctions and disjunctions in the heads of rules, it does understand choice formulas as heads, with the superscript ^{c} dropped. For instance, we can write

```
{p,q}
```

for

$$(p \vee \neg p) \wedge (q \vee \neg q)$$

and

```
{p} :- q
```

for

$$p \vee \neg p \leftarrow q.$$

Problem 3.2 (a) Consider the program

$$\begin{array}{l} \{p,q,r\}^c \\ s \leftarrow p, not\ q. \end{array}$$

What do you think its answer sets are? Use SMODELS to verify your conjecture. (b) Do the same for the program

$$\begin{array}{l} \{p,q\}^c \\ \{r,s\}^c \leftarrow not\ p. \end{array}$$

If the head of a rule is a choice formula Z^c for a large set Z then it may be possible to represent the rule in the language of LPARSE concisely using variables (see Section 1.5 on the use of variables in LPARSE). For instance, the rule

$$\{p_1,\ldots,p_7\}^c \leftarrow q \tag{14}$$

can be encoded as follows:

```
index(1..7).
{p(I) : index(I)} :- q.
```

Note how the "local" use of I in this example differs from the "global" use of variables in Section 1.5 and in the following example:

```
index(1..7).
{p(I)} :- q, index(I).
```

The last two lines correspond to the set of 7 rules:

$$\{p_i\}^c \leftarrow q \qquad (1 \leq i \leq 7). \tag{15}$$

In this example, the difference between local and global variables is not essential, however: the theorem on intuitionistic equivalence (Section 2.4) shows that replacing (14) with (15) in any program does not affect the program's answer sets.

3.2 Constraints

Problem 3.3 A set of atoms is an answer set of $F \wedge \neg G$ iff it is an answer set of F and does not satisfy G.

We observed in Section 2.3 that a conjunction $F \wedge G$ may have answer sets that are not found among the answer sets of F. The assertion of Problem 3.3 shows, however, that this cannot happen if the second conjunctive term begins with negation. Conjoining a formula with $\neg G$ leads only to the loss of answer sets—of all those that do not satisfy G.

Formulas beginning with negation are called *constraints*. In the language of LPARSE, constraints are written as rules with the empty head. For instance, constraint (8) is represented in LPARSE as

```
:- p.
```

Problem 3.4 Find the answer sets of the program

$$\begin{array}{l} \{p, q, r\}^c \\ \leftarrow p, q, not\ r \end{array}$$

(a) using properties of choice formulas and constraints; (b) using SMODELS.

We will now give an example of the use of answer set programming. How can SMODELS help us solve the satisfiability problem for a given set of clauses?[2] This can be accomplished by running SMODELS on the input consisting of

- a choice rule that allows the atoms to be assigned arbitrary truth values, and
- constraints, one per clause, eliminating the truth assignments that do not satisfy the given clauses.

For instance, the answer sets of the program from Problem 3.4 are the sets satisfying the clause $\neg p \vee \neg q \vee r$. The clauses

$$\begin{array}{l} p_1 \vee p_2,\ \ p_2 \vee p_3, \dots,\ p_{n-1} \vee p_n, \\ \neg p_1 \vee \neg p_2,\ \neg p_2 \vee \neg p_3,\ \dots, \neg p_{n-1} \vee \neg p_n \end{array}$$

can be represented by the program

$$\begin{array}{ll} \{p_1, \dots, p_n\}^c & \\ \leftarrow not\ p_i, not\ p_{i+1} & (1 \le i < n) \\ \leftarrow p_i, p_{i+1} & (1 \le i < n) \end{array}$$

and by the LPARSE file

[2] A *clause* is a disjunction of literals.

```
index(1..n).

{p(I) : index(I)}.

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

(The expression `I<n` in the last two rules instructs LPARSE not to substitute the value `n` for `I` in the process of grounding.)

System SMODELS computes the answer sets of this program very quickly even for relatively large values of n, say 1000. (Try it!) The computation does not involve, of course, generating each of the 2^n answer sets of the choice rule and verifying whether it satisfies the constraints; the algorithm used by SMODELS is more sophisticated and efficient.

3.3 Cardinality Expressions

Constraints used in ASP programs often involve conditions on the cardinality of a set of atoms. We will introduce special notation for such formulas.

For any finite set Z of literals and any nonnegative integer l ("lower bound"), by $l \leq Z$ we denote the disjunction of the formulas $\bigwedge_{L \in Y} L$ over all l-element subsets Y of Z. For instance,

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

stands for

$$(p \wedge q) \vee (p \wedge r) \vee (q \wedge r).$$

Clearly, if the elements of Z are atoms then a subset X of Z satisfies $l \leq Z$ iff the cardinality of X is at least l.

By $Z \leq u$, where u is a nonnegative integer ("upper bound"), we denote the formula $\neg(u + 1 \leq Z)$. Finally, $l \leq Z \leq u$ stands for

$$(l \leq Z) \wedge (Z \leq u).$$

As an example of the use of cardinaliy expressions in constraints, consider the program

$$\begin{array}{l} \{p_1, \ldots, p_n\}^c \\ \leftarrow \{p_1, \ldots, p_n\} \leq 0 \\ \leftarrow 2 \leq \{p_1, \ldots, p_n\}. \end{array} \tag{16}$$

The constraints eliminate two kinds of sets from the collection of answer sets of the choice rule: the empty set and the sets that have at least 2 elements. Consequently, the answer sets of (16) are the singletons $\{p_1\}, \ldots, \{p_n\}$.

The language of LPARSE allows us to use cardinality expressions

$$l \le Z,\ Z \le u,\ l \le Z \le u$$

in the bodies of rules, with the sign $\le$ dropped. For instance, program (16) can be written as

```
index(1..n).

{p(I) : index(I)}.

:- {p(I) : index(I)} 0.
:-  2 {p(I) : index(I)}.
```

Problem 3.5 Although the syntax of LPARSE does not allow us, generally, to use nested negations, the formula $\neg\neg p$ can be written as a cardinality expression that LPARSE understands. Find such an expression and use SMODELS to find the answer sets for the second of formulas (13).

If Z is a finite set of atoms, we will use

$$\begin{array}{rll} l \le Z^c & \text{as shorthand for} & Z^c \wedge (l \le Z), \\ Z^c \le u & \text{as shorthand for} & Z^c \wedge (Z \le u), \\ l \le Z^c \le u & \text{as shorthand for} & Z^c \wedge (l \le Z \le u). \end{array}$$

Problem 3.6 For any finite set Z of atoms,

$$\begin{array}{rll} l \le Z^c & \text{is intuitionistically equivalent to} & Z^c \wedge \neg(Z \le l-1), \\ Z^c \le u & \text{is intuitionistically equivalent to} & Z^c \wedge \neg(u+1 \le Z), \\ l \le Z^c \le u & \text{is intuitionistically equivalent to} & Z^c \wedge \neg(Z \le l-1) \wedge \neg(u+1 \le Z) \end{array}$$

$(l > 0)$.

In view of the theorem on intuitionistic equivalence (Section 2.4), the last assertion of Problem 3.6 shows that program (16) can be rewriten as

$$1 \le \{p_1, \dots, p_n\}^c \le 1. \tag{17}$$

The language of LPARSE allows us to use expressions

$$l \le Z^c,\ Z^c \le u,\ l \le Z^c \le u$$

in heads of rules, with both $\le$ and ^{c} dropped. For instance, program (17) can be written as

```
index(1..n).

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

Note that LPARSE understands the expression

```
l {...} u
```

in different ways depending on whether it occurs in the body or in the head of a rule: it stands for

$$l \leq \{\cdots\} \leq u$$

in the body, and for

$$l \leq \{\cdots\}^c \leq u$$

in the head.

Problem 3.7 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 Appendix: Propositional Logic

A.1 Syntax and Semantics

The definitions below are different from what is found in many textbooks on logic in two ways. First, negation is not treated here as a primitive; we define it in terms of implication and the 0-place connective $\bot$. Second, we identify truth-valued functions on the set of atoms with subsets of that set. For instance, instead of the function that maps the atoms p and r to t and all other atoms to f, we will talk about the set $\{p, r\}$.

(Propositional) formulas are formed from propositional atoms and the 0-place connective $\bot$ using the binary connectives $\wedge$, $\vee$ and $\rightarrow$. We use

$$\begin{array}{rll} \top & \text{as shorthand for} & \bot \rightarrow \bot, \\ \neg F & \text{as shorthand for} & F \rightarrow \bot, \\ F \leftrightarrow G & \text{as shorthand for} & (F \rightarrow G) \wedge (G \rightarrow F). \end{array}$$

Atoms and negated atoms are called *literals*.

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

- for an atom A, $X \models A$ if $A \in X$;
- $X \not\models \bot$;
- $X \models F \wedge G$ if $X \models F$ and $X \models G$;
- $X \models F \vee G$ if $X \models F$ or $X \models G$;
- $X \models F \rightarrow G$ if $X \not\models F$ or $X \models G$.

A formula F is a *tautology* if, for every set X of atoms, $X \models F$. A formula F is *equivalent* to a formula G if $F \leftrightarrow G$ is a tautology.

An occurrence of an atom A in a formula F is *positive* if the number of implications containing that occurrence in the antecedent is even, and *negative* otherwise. For instance, both occurrences of p in the formula

$$((p \rightarrow q) \wedge r) \rightarrow p$$

are positive, and q, r are negative. The following fact explains why this definition is useful.

Monotonicity Theorem. *If all occurrences of an atom A in a formula F are positive and $X \models F$ then $X \cup \{A\} \models F$. If all occurrences of an atom A in a formula F are negative and $X \not\models F$ then $X \cup \{A\} \not\models F$.*

A.2 Natural Deduction

In the natural deduction system $\mathbf{N}$, the derivable objects are *sequents*—expressions of the form $\Gamma \Rightarrow F$ ("F under the assumptions Γ"), where F is a formula and Γ is a finite set of formulas. Notationally, we will identify the set of assumption in a sequent with the list of its elements. For instance, we will write

$$\begin{array}{lll} F, G \Rightarrow H & \text{for} & \{F, G\} \Rightarrow H, \\ \Gamma, F \Rightarrow G & \text{for} & \Gamma \cup \{F\} \Rightarrow G, \\ \Gamma, \Delta \Rightarrow F & \text{for} & \Gamma \cup \Delta \Rightarrow G. \end{array}$$

The axiom schemas of $\mathbf{N}$ are

$$F \Rightarrow F \tag{18}$$

and

$$\Rightarrow F \vee \neg F. \tag{19}$$

The latter is called *the law of excluded middle.* The inference rules of $\mathbf{N}$ are:

$$(\wedge I)\ \frac{\Gamma \Rightarrow F \quad \Delta \Rightarrow G}{\Gamma, \Delta \Rightarrow F \wedge G} \qquad (\wedge E)\ \frac{\Gamma \Rightarrow F \wedge G}{\Gamma \Rightarrow F} \quad \frac{\Gamma \Rightarrow F \wedge G}{\Gamma \Rightarrow G}$$

$$(\vee I)\ \frac{\Gamma \Rightarrow F}{\Gamma \Rightarrow F \vee G} \quad \frac{\Gamma \Rightarrow G}{\Gamma \Rightarrow F \vee G} \qquad (\vee E)\ \frac{\Gamma \Rightarrow F \vee G \quad \Delta_1, F \Rightarrow H \quad \Delta_2, G \Rightarrow H}{\Gamma, \Delta_1, \Delta_2 \Rightarrow H}$$

$$(\to I)\ \frac{\Gamma, F \Rightarrow G}{\Gamma \Rightarrow F \to G} \qquad (\to E)\ \frac{\Gamma \Rightarrow F \quad \Delta \Rightarrow F \to G}{\Gamma, \Delta \Rightarrow G}$$

$$(C)\ \frac{\Gamma \Rightarrow \bot}{\Gamma \Rightarrow F}$$

$$(W)\ \frac{\Gamma \Rightarrow F}{\Gamma' \Rightarrow F}$$

if $\Gamma \subseteq \Gamma'$

Among the first six inference rules, the rules in the left column are *introduction rules*, and the rules in the right column are *elimination rules*. Rule (C) is the *contradiction rule*, and (W) is *weakening*.

Since we defined $\neg F$ as an abbreviation for $F \to \bot$ (Section A.1), "negation introduction"

$$\frac{\Gamma, F \Rightarrow \bot}{\Gamma \Rightarrow \neg F}$$

is a special case of $(\to I)$, and "negation elimination"

$$\frac{\Gamma \Rightarrow F \quad \Delta \Rightarrow \neg F}{\Gamma, \Delta \Rightarrow \bot}$$

is a special case of $(\to E)$. Similarly, the introduction and elimination rules for equivalence

$$\frac{\Gamma \Rightarrow F \to G \quad \Delta \Rightarrow G \to F}{\Gamma, \Delta \Rightarrow F \leftrightarrow G} \qquad \frac{\Gamma \Rightarrow F \leftrightarrow G}{\Gamma \Rightarrow F \to G} \qquad \frac{\Gamma \Rightarrow F \leftrightarrow G}{\Gamma \Rightarrow G \to F}$$

are special cases of $(\wedge I)$ and $(\wedge E)$.

To prove a formula F in system **N** means to prove the sequent $\Rightarrow F$. For instance, here is a proof of the formula

$$\neg\neg F \leftrightarrow F \tag{20}$$

in **N**:

$$\begin{array}{rll}
1. & F \Rightarrow F & \text{— axiom} \\
2. & \neg F \Rightarrow \neg F & \text{— axiom} \\
3. & F, \neg F \Rightarrow \bot & \text{— by } (\to E) \text{ from 1, 2} \\
4. & F \Rightarrow \neg\neg F & \text{— by } (\to I) \text{ from 3} \\
5. & \Rightarrow F \to \neg\neg F & \text{— by } (\to I) \text{ from 4} \\
6. & \neg\neg F \Rightarrow \neg\neg F & \text{— axiom} \\
7. & \neg F, \neg\neg F \Rightarrow \bot & \text{— by } (\to E) \text{ from 2, 6} \\
8. & \neg F, \neg\neg F \Rightarrow F & \text{— by } (C) \text{ from 7} \\
9. & \Rightarrow F \vee \neg F & \text{— axiom} \\
10. & \neg\neg F \Rightarrow F & \text{— by } (\vee E) \text{ from 9, 1, 8} \\
11. & \Rightarrow \neg\neg F \to F & \text{— by } (\to I) \text{ from 10} \\
12. & \Rightarrow \neg\neg F \leftrightarrow F & \text{— by } (\wedge I) \text{ from 11, 5}
\end{array}$$

Soundness and Completeness Theorem for System N. *A formula F is provable in* **N** *iff F is a tautology.*

A.3 Intuitionistic Provability

A sequent or a formula is *intuitionistically provable* if it can be proved in system **N** without axiom schema (19). A formula F is *intuitionistically equivalent* to a formula G if $F \leftrightarrow G$ is intuitionistically provable.

For instance, every formula of the form $F \to \neg\neg F$ is intuitionistically provable, because the first 5 lines of the proof of (20) given in Section A.2 don't contain references to the law of excluded middle. That axiom schema is used, however, in line 9 of the proof, so that we have no grounds to assert that the implication in the opposite direction is intuitionistically provable. Actually, the implication from $\neg\neg F$ to F is known to be *not* provable without the law of excluded middle in many cases (for instance, when F is an atom). In other words, a formula, generally, is not intuitionistically equivalent to its double negation.

Here is another example of the difference between equivalence and intuitionistic equivalence. It's easy to check that the formulas

$$\neg(F \wedge G), \quad F \to \neg G, \quad G \to \neg F \tag{21}$$

are intuitionistically equivalent to each other. But it is known that the formulas

$$F \vee G, \quad \neg F \to G, \quad \neg G \to F \tag{22}$$

are not intuitionistically equivalent to each other if, for instance, F and G are two different atoms.

Intuitionistic provability, which plays an important role in the theory of answer sets, was introduced in [Heyting, 1930], long before the invention of logic programming.

We will now state two theorems that relate intuitionistic provability to the concept of a tautology.

Glivenko's Theorem [Glivenko, 1929]. A formula of the form $\neg F$ is intuitionistically provable iff it is a tautology.

If Z is a finite set of atoms, by Z^c we will denote the conjunction of the excluded middle formulas for all atoms in Z:

$$\bigwedge_{A \in Z} (A \vee \neg A).$$

Theorem on Excluded Middle. *Let $A_1, \dots, A_n$ be all atoms occurring in a formula F. The sequent*

$$\{A_1, \dots, A_n\}^c \Rightarrow F$$

is intuitionistically provable iff F is a tautology.

For instance, the sequent

$$p \vee \neg p \Rightarrow \neg\neg p \leftrightarrow p$$

is intuitionistically provable because $\neg\neg p \leftrightarrow p$ is a tautology.

Replacement Theorem for Intuitionistic Logic. *Let F, G, F', G' be formulas such that G' is obtained from F' by replacing an occurrence of F with G. (a) The sequent*

$$F \leftrightarrow G \Rightarrow F' \leftrightarrow G'$$

is intuitionistically provable. (b) If F is intuitionistically equivalent to G then F' is intuitionistically equivalent to G'.

The replacement theorem shows, for instance, that replacing a subformula of one of the forms (21) in any formula with another expression from that list will produce an intuitionistically equivalent result.

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.

[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.

[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.

[Ferraris, 2005] Paolo Ferraris. Answer sets for propositional theories.[3] Unpublished draft, 2005.

[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.

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

[Glivenko, 1929] Vladimir Glivenko. Sur quelques points de la logique de M. Brouwer. *Académie Royale de Belgique. Bulletins de la Classe des Sciences, se'rie 5*, 15:183–188, 1929.

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

[Lifschitz and Woo, 1992] Vladimir Lifschitz and Thomas Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proc. Third Int'l*

[3] `http://www.cs.utexas.edu/users/otto/papers/proptheories.ps` .

Conf. on Principles of Knowledge Representation and Reasoning, pages 603–614, 1992.

[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.

[McCarthy, 1980] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39,171–172, 1980. Reproduced in [McCarthy, 1990].

[McCarthy, 1990] John McCarthy. *Formalizing Common Sense: Papers by John McCarthy.* Ablex, Norwood, NJ, 1990.

[McDermott and Doyle, 1980] Drew McDermott and Jon Doyle. Nonmonotonic logic I. *Artificial Intelligence*, 13:41–72, 1980.

[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-Verlag, 1997.

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

[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.