CSE 591 Knowledge Representation and Reasoning, Fall 2009

2. Introduction to Answer Set Programming

Beyond the study of proofs in mathematics, logic has been applied to designing and reasoning about computer hardware and software. One of prominent applications of logic in computer science is the use of logic as a programming language. *Logic programming* specifies *what* is to be computed, but not necessarily *how* it is computed.

> A logic program is a set of axioms, or rules, defining relations between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have the desired meaning. (From *The Art of Prolog*, page 9.)

In this handout, we will study "Answer Set Programming (ASP)," a form of declarative programming oriented towards difficult combinatorial search problems. It has been applied, for instance, to knowledge representation, planning, and product configuration problems in artificial intelligence and to graph-theoretic problems arising in VLSI design, historical linguistics, and bioinformatics.

The idea of ASP is to represent the search problem we are interested in as the problem of finding the answer sets of a formula, and then find the solutions using *answer set solvers*, such as the systems SMODELS, DLV, ASSAT, CMODELS, and CLINGO.

Traditional Answer Set Programs

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 positive programs, a.k.a. Horn programs.

Syntax

A *positive* rule has the form

$$A_0 \leftarrow A_1 \wedge \cdots \wedge A_m \tag{1}$$

where $m \geq 0$ and $A_0, \dots, A_m$ are propositional atoms. We identify (1) with implication

$$A_1 \wedge \cdots \wedge A_m \rightarrow A_0. \tag{2}$$

The consequent A_0 of implication (1) is often called the *head* of the rule, and the antecedent

$$A_1 \wedge \cdots \wedge A_m$$

is often called the *body* of the rule. If the body is the empty conjunction ($m = 0$) then the rule is called a *fact* and identified with its head A_0.

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

$$\begin{array}{l} p \\ r \leftarrow p \wedge q \end{array} \tag{3}$$

is a positive program. We identify a positive program with the conjunction of implications. For instance (3) is identified with

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

Answer Sets of a Positive Program

Below we identify an interpretation with the set of atoms that are true in it. For instance, an interpretation I of signature $\{p, q\}$ such that $I(p) = \mathbf{f}$, and $I(q) = \mathbf{t}$ is identified with $\{q\}$.

2.1 For any positive program Π, the intersection of all sets satisfying Π satisfies Π also.

We call the *minimal* set of atoms that satisfy Π the *answer set* of Π. For instance, the sets of atoms satisfying program (3) are

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

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

Intuitively, we can think of (1) as a rule for generating atoms: once you have generated $A_1, \dots, A_m$, 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 $\{p, q, r\}$.

Answer Sets of a Traditional Program with Negation

A *traditional rule* extends the syntax of a positive rule as in the following:

$$A_0 \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n \tag{4}$$

where $n \geq m \geq 0$ and $A_0, \ldots, A_n$ are propositional atoms. Similarly as before, we call A_0 the *head*, and

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

the *body* of the rule.

A *traditional program* is a finite set of traditional rules. For instance, the following is a traditional program that is not positive.

$$\begin{aligned} p &\leftarrow \neg q \\ q &\leftarrow \neg r \end{aligned} \tag{5}$$

To extend the definition of an answer set to traditional programs, we will introduce an auxiliary definition. The *reduct* Π^X of a traditional program Π relative to a set X of atoms is obtained from Π by replacing every occurrence of the form $\neg A$ by $\bot$ if $X \models A$ and $\top$ otherwise. Note that Π^X can be equivalently rewritten as a positive program by first dropping all rules that contains $\bot$ and then removing all occurrences of $\top$ from the remaining rules. Thus Π^X is essentially a transformation that turns a traditional program into a positive program.

We say that X is an *answer set* of Π if X is the answer set of Π^X (that is, the minimal 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 the previous section: for any positive traditional program Π, X is the minimal set of atoms satisfying Π^X iff X is the minimal set of atoms satisfying Π.

Example: if Π is (5) and $X = \{q\}$ then the reduct Π^X is

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

which is equivalent to the single fact q. The answer set of Π^X is $\{q\}$. Consequently, $\{q\}$ is an answer set of Π.

Intuitively, rule (4) 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. 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.

2.2^e Check that $\{p\}$ is not an answer set of program (5).

Does program (5) have answer sets other than $\{q\}$? We can find this out by checking each of the remaining 6 subsets of $\{p, q, r\}$. We can do a little better by using the following general properties of answer sets of traditional programs.

2.3 Prove that 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.

2.4 Prove that 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 (5), the assertion of Problem 2.3 tells us that its answer sets do not contain r, so that we only need to check the subsets of $\{p, q\}$. By the assertion of Problem 2.4, $\emptyset$ 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 (5).

2.5 Find an answer set of the program Π_n consisting of n rules

$$p_i \leftarrow \neg p_{i+1} \qquad (1 \leq i \leq n)$$

where n is a positive integer. (Program (5) is essentially Π_2.)

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

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

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

$$r \leftarrow \neg r \tag{7}$$

has no answer sets.

2.6 Find all answer sets of the following program, which extends (6) by two additional rules:

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

2.7 Find all answer sets of the following combination of programs (6) and (7) plus one more rule:

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

GRINGO and CLASP

There are several answer set solvers available today. For this class we are going to use GRINGO (grounder) and CLASP (solver), developed at the University of Potsdam in Germany. They can be be downloaded from

`http://potassco.sourceforge.net/` .

along with the user's guide. The input language of GRINGO is a superset of the language of LPARSE, which has been designed and implemented at the Helsinki University of Technology in Finland.

In the input language of GRINGO, as in Prolog, `:-` stands for $\leftarrow$, `not` stands for $\neg$, **,** stands for $\wedge$, **;** stands for $\vee$, and each rule is followed by a period. If we want to find, for instance, the answer sets of the program from Problem 2.6, we create the file

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

called, say, `p2.6` . Then we invoke CLINGO as follows:

```
% glingo p2.6 | clasp 0
```

The zero at the end indicates that we want to compute all answer sets; a positive number k would tell CLINGO 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
p r
Answer: 2
q r

Models      : 2
Time        : 0.000
```

A group of rules that follow a pattern can be often described concisely in the input language of GRINGO using schematic variables. As in Prolog, variables must be capitalized. Consider, for instance, the programs Π_n from Problem 2.5. 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 GRINGO.

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 $(n = 1, 2, \dots)$ can be described by a program schema with the parameter n:

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

Let's name this file `p2.5`. When this schema is given to GRINGO as input, the value of the constant n can be specified in the command line using the option `-c`, as follows:

```
% gringo -c n=7 p2.5 | clasp 0
```

When the input file uses domain predicates, the output of GRINGO 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 by including the following line in the input file:

```
#hide index/1.
```

In order to selectively include the atoms, one may use the `#show` declarative instead. Typically one hide all predicates via

```
#hide.
```

and selectively show atoms of certain predicates `p/n` in the output via "`#show p/n`".

2.8^e Use CLASP to verify that Π_n has a unique answer set for $n = 10$ and $n = 100$.

2.9^e Consider the program obtained from Π_n by adding the rule

$$p_{n+1} \leftarrow \neg p_1.$$

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

Later we will discuss several other useful features of the input language of GRINGO.

Modern Answer Set Programs

A *rule* is an implication of the form

$$F \leftarrow G \tag{8}$$

where F and G are formulas that contain no connectives other than $\{\bot, \top, \wedge, \vee, \neg\}$. F is called the *head* of the rule and G is called the *body*. If G is $\top$ we often identify the rule with its head by dropping $\leftarrow \top$. If F is $\bot$ we often write the rule as $\leftarrow G$.

A *program* is a finite set of rules.

The answer sets of a program are defined as follows. The *reduct* Π^X of a program Π relative to a set X of atoms is obtained from Π by replacing all maximal occurrences of $\neg H$ by

- $\top$ if $X \models \neg H$,
- $\bot$ otherwise.

We say that X is an *answer set* of Π if X is a minimal set satisfying Π^X. As before the minimality of X is understood here in the sense of set inclusion.

2.10 (a) Program $\neg\neg p$ has no answer sets. (b) Each of the two programs

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

and

$$p \leftarrow \neg\neg p \tag{10}$$

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

The language of GRINGO is in fact more restricted than the syntax above. A rule allowed in that language is an extension of a traditional rule described in the previous handout. We will discuss such extensions in this handout.

Nonmonotonicity of Answer Set Semantics

It is easy to see that any answer set of program Π satisfies Π. (Try to prove it!) In this sense, the answer set semantics is stronger than the propositional logic semantics, which is given by the concept of satisfaction. The sets satisfying a formula are 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 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].

2.11

(a) Program

$$p \vee q$$

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

(b) Program

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

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

These facts provide one more example of nonmonotonicity: after appending the last two rules in (b), the program $p \vee q$ 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 last two rules to the disjunction makes $\{p, q\}$ an answer set; the disjunction "becomes inclusive."

Choice Formulas 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. A "choice rule" is a program with many answer sets that are more than the collection of sets that we want to describe. The additional answer sets can be removed by adding "constraints" to this program.

Choice Formulas

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

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

2.12 Prove the following statement:

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 GRINGO 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

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

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 GRINGO concisely using variables. For instance, the rule

$$\{p_1, \dots, p_7\}^c \leftarrow q \qquad (11)$$

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 the following example:

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

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

$$\{p_i\}^c \leftarrow q \qquad (1 \le i \le 7). \tag{12}$$

In this example, the difference between local and global variables is not essential, however: there is a theory that tells that replacing (11) with (12) in any program does not affect the program's answer sets.[1]

Constraints

A *constraint* is a rule with the head $\bot$, that is to say, a rule of the form $\bot \leftarrow F$. (Recall it can be abbreviated as $\leftarrow F$.) The following theorem tells us how adding a constraint to a program affects the collection of its answer sets.

Theorem on Constraints [Lifschitz *et al.*, 1999]. For any program Π and formula F, a set X of atoms is an answer set for $\Pi \cup \{\leftarrow F\}$ iff X is an answer set for Π and does not satisfy F.

2.13^e Find the answer sets of the program

$$\begin{array}{l} \{p, q, r\}^c \\ \leftarrow p, q, \neg r \end{array}$$

(a) using Theorem on Constraints; (b) using CLASP.

We observed earlier that a program $\Pi_1 \cup \Pi_2$ may have answer sets that are not found among the answer sets of Π_1. The assertion of the Theorem on Constraints shows, however, that this cannot happen if Π_2 is a constraint. Conjoining a program with a constraint leads only to the loss of answer sets.

2.14 Prove Theorem on Constraints.

The combination of choice rules and constraints yields a way to embed propositional logic into the answer set semantics as follows.

2.15^e (a) For any propositional formula F, a set X of atoms occurring in F is a model of F iff X is an answer set of $Z^c \wedge \neg\neg F$. (b) The statement (a) remains true if $\neg\neg F$ in it is replaced with F.

[1]This theory is called "strong equivalence," which is beyond the scope of this class.

This fact allows us to compute the models of propositional formulas using answer set solvers. However, current answer set solvers requires the input to be in rule forms, which can be viewed as a conjunction of certain kinds of implications.

2.16 Use CLASP to find all models of each of the following formulas.

(a) $(p_1 \wedge q_1) \vee (p_2 \wedge q_2) \vee (p_3 \wedge q_3)$

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

Cardinality Formulas

Constraints used in ASP programs often involve conditions on the cardinality of a set of atoms. We will introduce special notation for such formulas. A *cardinality atom* is an expression of the form

$$l\{A_1, ..., A_n\} \tag{13}$$

where l is a nonnegative integer ("bound") and all A_i are atoms. The concept of *cardinality formula* extends the definition of a formula in propositional logic as follows.

- every atom is a cardinality formula;
- every cardinality atom is a cardinality formula;
- both 0-place connectives are cardinality formulas;
- if F is a cardinality formula, then $\neg F$ is a cardinality formula;
- if F and G are cardinality formulas, then $(F \wedge G)$ and $(F \vee G)$ are cardinality formulas.

Similarly we extend rules and programs by referring to cardinality formulas in place of formulas. That is, a *cardinality rule* is an expression of the form (8) where F and G are cardinality formulas. A *cardinality program* is a finite set of cardinality rules.

The concept of satisfaction for cardinality formulas is defined by extending the concept of satisfaction in propositional logic. We say that a set X of atoms satisfies (13) if the number of atoms among $A_1, ..., A_n$ that belong to X is at least l. Otherwise the definition remains the same.

The definition of a reduct remains the same except that we refer to the extended notion of satisfaction instead.

2.17 Find all answer sets of the following programs.

(a)

$$\begin{array}{l}\{p,q,r\}^c\\ s \leftarrow 2\,\{p,q,r\}.\end{array}$$

(b)

$$\begin{array}{l}\{p,q,r\}^c\\ p \leftarrow 2\,\{p,q,r\}.\end{array}$$

Notation:

$$\{A_1,\dots,A_n\}\,u$$

stands for

$$\neg(u+1)\,\{A_1,\dots,A_n\};$$

$$l\,\{A_1,\dots,A_n\}\,u$$

stands for

$$l\,\{A_1,\dots,A_n\}\ \wedge\ \{A_1,\dots,A_n\}\,u$$

As an example of the use of cardinality atoms in constraints, consider the program

$$\begin{array}{l}\{p_1,\dots,p_n\}^c\\ \leftarrow \{p_1,\dots,p_n\}\,0\\ \leftarrow 2\,\{p_1,\dots,p_n\}.\end{array} \tag{14}$$

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 (14) are the singletons $\{p_1\},\dots,\{p_n\}$.

Program (14) can be written in the language of GRINGO as

```
index(1..n).

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

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

2.18 Although the syntax of GRINGO does not allow us, generally, to use nested negations, the occurrence of formula $\neg\neg p$ can sometimes be written as a cardinality formula that GRINGO understands. Find such a formula and use CLASP to find the answer sets

$$p \leftarrow \neg\neg p. \tag{15}$$

(Do not use choice rules.)

Notation:

$$\begin{array}{rll} l\,\{A_1,\dots,A_n\}^c & \text{stands for} & \{A_1,\dots,A_n\}^c \;\wedge\; l\,\{A_1,\dots,A_n\}, \\ \{A_1,\dots,A_n\}^c\,u & \text{stands for} & \{A_1,\dots,A_n\}^c \;\wedge\; \{A_1,\dots,A_n\}\,u, \\ l\,\{A_1,\dots,A_n\}^c\,u & \text{stands for} & \{A_1,\dots,A_n\}^c \wedge\; l\,\{A_1,\dots,A_n\}\,u. \end{array}$$

Proposition on cardinality atom *For any pairwise distinct atoms $A_1,\dots,A_n$, nonnegative integers l and u, and a set X of atoms,*

(i) X is an answer set of $l\,\{A_1,\dots,A_n\}^c$ iff $X \subseteq \{A_1,\dots,A_n\}$ and $l \leq |X|$;

(ii) X is an answer set of $\{A_1,\dots,A_n\}^c\,u$ iff $X \subseteq \{A_1,\dots,A_n\}$ and $|X| \leq u$;

(iii) X is an answer set of $l\,\{A_1,\dots,A_n\}^c\,u$ iff $X \subseteq \{A_1,\dots,A_n\}$ and $l \leq |X| \leq u$.

The language of GRINGO allows us to use expressions

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

in heads of rules, with ^{c} dropped. For instance, program

$$1\,\{p_1,\dots,p_n\}^c\,1. \tag{16}$$

can be written as

```
index(1..n).

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

Note that GRINGO 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

$$l\,\{\cdots\}\,u$$

in the body, and for

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

in the head.

2.19 Consider the program

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

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

References

[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.

[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.

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

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