3. 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 $A_0, \ldots, A_m$ 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 $\{q\}$.

3.1 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 $A_1, \ldots, A_m$, 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 $\{p, q, r\}$.

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.

$$\begin{array}{l} p \leftarrow \neg q \\ q \leftarrow \neg r \end{array} \tag{5}$$

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 $\top$ if $X \models \neg A$ and $\bot$ 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

$$p \leftarrow \bot$$
$$q \leftarrow \top,$$

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 $A_1, \ldots, A_m$ *provided that none of the atoms* $A_{m+1}, \ldots, A_n$ *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.

3.2^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.

3.3 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.

3.4 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 3.3 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 3.4, $\emptyset$ 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).

3.5 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.

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

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

3.7 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 3.6, we create the file

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

called, say, `p3.6` . Then we invoke CLASP as follows:

```
% glingo p3.6 | clasp 0
```

The zero at the end indicates that we want to compute all answer sets; a positive number k would tell CLASP 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 3.5. 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 `p3.5`. 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 p3.5 | 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`".

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

3.9^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.

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.

[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.