KNOWLEDGE REPRESENTATION, REASONING AND DECLARATIVE PROBLEM SOLVING

By Jicheng Zhao
based on the notes of others

Class 7: Declarative Problem Solving and $AnsProlog^{\neg}$

Forall and Exists

Example 1 *write an implementation in AnsProlog for the following rule:*
Refuse admission to an applicant if he/she has not taken any (at least one) honors class.
We were given the following predicates to use: $refuse\_admission(X)$, $applicant(X)$, *and* $taken\_honor\_class(X, Y)$.

One attempt:

$refuse\_admission(X) \leftarrow applicant(X), \textbf{not}\; taken\_honor\_class(X, Y), course(Y).$

However, this rule fails in the following instance:

$$applicant(john).$$
$$applicant(george).$$
$$taken\_honor\_class(john, chemistry).$$
$$taken\_honor\_class(george, physics).$$
$$course(chemistry).$$
$$course(physics).$$

The rule can be grounded to:

$refuse(john) \leftarrow$
$applicant(john), \textbf{not}\; taken\_honor\_class(john, chemistry), course(chemistry).$
$refuse(john) \leftarrow$
$applicant(john), \textbf{not}\; taken\_honor\_class(john, physics), course(physics).$
$refuse(george) \leftarrow$

$applicant(george), \textbf{not } taken\_honor\_class(george, chemistry), course(chemistry$
$refuse(george) \leftarrow$
$applicant(george), \textbf{not } taken\_honor\_class(george, physics), course(physics).$
In the two applications of the middle two rules, the applicant will be rejected because there exists an honors class which that individual did not take. This answer would have been correct if the problem was to reject an applicant who had not taken **ALL** honors classes.
When using variables with **not**s we have to be careful. If there *exists* a Y such that the condition is not true, the rule will be applied.
For example, in the rules
$parent(X) \leftarrow f(Y, X).$
$parent(X) \leftarrow m(Y, X).$ if there *exists* a Y, then X is a parent. X is a parent if there exists a Y such that X is a f of Y or X is a m of Y. Any variable that doesn't appear in the head acts as a "there exists."
The correct answer to the problem is:
$taken\_an\_honor\_class(X) \leftarrow taken\_honor\_class(X, Y).$
$refuse\_admission(X) \leftarrow$

$$applicant(X), \mathbf{not}\ taken\_an\_honor\_class(X).$$

$\neg$ notation

Example 2 *Consider the following program:*
$fly(X) \leftarrow bird(X), \textbf{not } ab(X).$
$bird(tweety).$
$ab(X) \leftarrow penguin(X).$
$bird(X) \leftarrow penguin(X).$
tweety is a bird. Since we cannot prove that he is abnormal, it follows that tweety flies. However, suppose we observe that he does not fly. How can we represent this?

1. How about adding:

$$\leftarrow fly(tweety).$$

   This rule doesn't work. The program has no stable models by adding it.

2. We can add:

$$abnormal(tweety).$$

   By adding this rule, we cannot prove that $fly(tweety)$. This works. But it is not good enough if we want to conclude "$\neg fly(tweety)$".

3. Similar to the previous method, one solution would be to write the first rule of the program with the following two rules:

$$\begin{aligned} fly(X) &\leftarrow bird(X), \mathbf{not}\neg fly(X). \\ \neg fly(X) &\leftarrow abnormal(X). \end{aligned}$$

   Then if we observe that tweety does not fly, we can simply add the fact:

$$\neg fly(tweety).$$

Another use for the $\neg$ notation is in the following example:

Example 3 Cross the track if no train is coming.

However, the rule

$$cross \leftarrow \textbf{not } train.$$

is dangerous because it means to cross if we cannot prove that a train is coming. We want to say

Do not cross the track unless we KNOW that no train is coming.

We can say

$$cross \leftarrow \neg train.$$

to mean that we cross if we are sure there is no train.

The point is, this notation is useful in writing programs, but it can always be eliminated by introducing a new atom

$$n\_fly(tweety).$$

and add one constraint:

$$\leftarrow n\_fly(X), fly(X).$$

AnsProlog$^{\neg}$

$AnsProlog^{\neg, -\mathbf{not}}$ rules are of the form

$$L_0 \leftarrow L_1, ..., L_m.$$

where the L_i's are literals, e.g. p or $\neg p$.

- The answer set of an $AnsProlog^{\neg, -\mathbf{not}}$ program is calculated in the same manner as $AnsProlog^{-\mathbf{not}}$. It is the minimal set of *atoms* which is *closed* under the rules.
- except that if this set has $\neg p$ and p for some atom p. In this case, then *lit* is the answer set.

$AnsProlog^{\neg}$ rules are of the form

$$L_0 \leftarrow L_1, ..., L_m, \mathbf{not} L_{m+1}, ..., \mathbf{not} L_n.$$

- The answer set of an $AnsProlog^{\neg}$ program is got in the same way as $AnsProlog$.

- except that if this set has $\neg p$ and p for some atom p. In this case, then the set of all atoms in the program is the answer set.

Example 4

$$\begin{aligned} \neg a &\leftarrow b. \\ b &\leftarrow . \\ a &\leftarrow . \end{aligned}$$

This program is not consistent. Its answer set is $\{a, b, \neg a, \neg b\}$. *It consists of all atoms in the program.*

Example 5 *Given program* Π*:*

$$\begin{aligned} \neg a &\leftarrow b. \\ b &\leftarrow . \end{aligned}$$

The answer set of Π *is* $S = \{\neg a, b\}$.

Let us consider another program π^+:

$$n\_a \leftarrow b.$$

$$b \leftarrow .$$

Its answer set is $S^+ = \{b, n\_a\}$.

We can draw a conclusion that if S is a **consistent** set of literals, then S is an answer set of Π iff S^+ is an answer set of Π^+. While Π^+ (S^+) is got from Π (S) by replacing all occurrences of $\neg p$ with a new atom n_p.

In other word, if the answer set of the program is consistent, then we can deal with it in the same way as a $AnsProlog$.

Example 6 *Consider the program*

$$\begin{aligned} a &\leftarrow b. \\ \neg a &\leftarrow . \\ b &\leftarrow . \end{aligned}$$

The answer set for the above program is $\{b, \neg a\, a, \neg b\}$.

Contrast this with the program

$$\begin{aligned} a &\leftarrow b. \\ n\_a &\leftarrow . \\ b &\leftarrow . \end{aligned}$$

whose answer set is $\{b, n\_a, a\}$.
The main point of this discussion is that **not** is false by default – we can't prove it true. But $\neg$ means that we *know* it's not true.

Example 7 *The program*

$$
\begin{aligned}
\neg a &\leftarrow \neg b. \\
\neg b &\leftarrow .
\end{aligned}
$$

has an answer set $\{\neg b, \neg a\}$. *Whereas the program*

$$
\begin{aligned}
b &\leftarrow a. \\
\neg b &\leftarrow .
\end{aligned}
$$

has answer set $\{\neg b\}$. a *is* unknown *with respect to the answer set. However, these two programs are equivalent in propositional logic.*

It is important to know that you don't know something. This allows us to say that we don't know the answer.

Example 8

$$
\begin{aligned}
eligible(X) &\leftarrow highGPA(X).\\
eligible(X) &\leftarrow special(X), fairGPA(X).\\
\neg eligible(X) &\leftarrow \neg special(X), \neg highGPA(X).\\
interview(X) &\leftarrow \mathbf{not}\ eligible(X), \mathbf{not}\ \neg eligible(X).\\
fairGPA(john).&\\
\neg special(john).&
\end{aligned}
$$

From these rules, we cannot determine if john is eligible or not. So we recommend that he be interviewed.

Closed world assumption and Open world assumption

With closed world assumptions, if something cannot be proven then it is considered false. With open world assumptions, if something cannot be proven then we simply do not know its truth value.
Consider the following program:

$$
\begin{aligned}
anc(X,Y) &\leftarrow par(X,Y). \\
anc(X,Y) &\leftarrow par(X,Z), anc(Z,Y). \\
par(a,b)&. \\
par(b,c)&. \\
par(d,e)&.
\end{aligned}
$$

We've defined what an ancestor is. Under *closed world assumptions*, any individual (a, b, c, d, or e) that is not shown to be an ancestor of one of the other individuals is by default not an ancestor of that individual. Remember, either something is provably true or it is assumed false.

However, under *open world assumptions*, this is not the case. The lack of proof that individual X is not individual Y's ancestor does not mean that individual X is not individual Y's ancestor. It simply means that we don't know. Now suppose we happen to know that individual X is, in fact, not individual Y's ancestor. How can this be represented using *open world assumptions*?

Suppose complete information is not available. We only know

$$\begin{aligned} par(a,b). \\ \neg par(b,c). \\ \neg par(a,d). \end{aligned} \tag{1}$$

Now, how can we define rules for "not ancestor".

1. We could say

$$\neg anc(X,Y) \leftarrow \textbf{not } anc(X,Y).$$

   This is incorrect indeed since there is something unknown. But we want to be **certain** that someone is not another's ancestor before we affirmatively make such a conclusion.

2. The rule

$$\neg anc(X, Y) \leftarrow anc(Y, X).$$

works for a small subset of instances but it is too narrow.

We will discuss this further in next class.