CSE571 SCRIBE NOTES

Andrew Davis

September 17, 2003

Our problem from the previous class was to try to 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:

$refuse\_admission(X)$
$applicant(X)$
$taken\_honor\_class(X, Y)$

One idea suggested in class was the rule:

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

However, this rule fails in the following instance:

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

The resulting Herbrand Base is:

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

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

$refuse(george) \leftarrow$
$applicant(george), \textbf{not}\ taken\_honor\_class(george, chemistry)$

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

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:

$taken\_an\_honor\_class(X) \leftarrow taken\_honor\_class(X, Y).$

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

The lesson here is, if a variable appears only after a **not**, and nowhere else, it means "there exists."

As a technical sidenote, in smodels we would need a domain specifier for the classes – i.e. we would need to say $class(chemistry)$ and $class(physics)$, and that predicate would need to be present in every rule that referenced a class.

This concept regarding variables and "there exists" is important for the current homework assignment. Any variable that appears only in the body of a rule acts as a "there exists."

$\neg$ Notation

Birds normally fly.

$fly(X) \leftarrow bird(X), \textbf{not}\ ab(X).$
$bird(tweety).$
$ab(X) \leftarrow penguin(X).$
$bird(X) \leftarrow penguin(X).$

If we want to say that tweety doesn't fly, we can say

$ab(tweety).$

We should be able to say

$\neg fly(tweety).$

The point is, this notation is useful in writing programs, but it can always be eliminated.

We could also say

$n\_fly(tweety).$

Another use for the ¬ notation is in the following example

Cross the track if no train is coming.

However, the rule

$cross \leftarrow$ **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.

Closed World Assumption

In the following example:

$anc(X, Y) \leftarrow par(X, Y).$
$anc(X, Y) \leftarrow par(X, Z), anc(Z, Y).$

$par(a, b).$
$par(b, c).$
$par(d, e).$

this is "closed world assumption." Everything we know about parents is here.

But suppose we found old DNA samples, so complete information is not available. Suppose we only know

$par(a, b).$
$\neg par(b, c).$
$\neg par(a, d).$

this is an open world assumption where some things are known and some are unknown. We can still use the ancestor rules from above, but we need rules for "not ancestor."

We could say

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

then from the above rules, we could conclude

$\neg anc(a, c).$

But we want to be **certain** that someone is not another's ancestor before we affirmatively make such a conclusion.

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

works for a small subset of instances. We want something that will work for all cases.

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

was another suggestion. But this won't work either.

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

would only work for only a small set of cases.

The solution to this problem is left as an exercise for the next class. Students are asked not to look in the book for the solution to it.

AnsProlog$^{\neg}$

There are two kinds of AnsProlog$^{\neg}$ – AnsProlog$^{\neg}$ and AnsProlog$^{\neg,-\mathbf{not}}$.
AnsProlog$^{\neg}$ rules are of the form

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

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 $\neg p$. These can be done in the same manner as AnsProlog$^{\neg,-\mathbf{not}}$ programs:

- minimal set of *literals* which is *closed* under the rules. (It was atoms instead of literals in the previous case.)
- except that if this set has $\neg p$ and p then *lit* is the answer set.

example:

$\neg a \leftarrow b.$
$b \leftarrow .$
$a \leftarrow .$

gives as its answer set $lit = \{a, b, \neg a, \neg b\}$. Lit means that all other possible literals get added (in this case, $\neg b$). The answer set consists of everything. This is not a consistent program.

For AnsProlog$^\neg$ programs, we do the same as before. Apply transformations to remove **not** from the rules.

An Example

The program π is:

$$\neg a \leftarrow b.$$

$$b \leftarrow .$$

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

For the program π^+

$$n\_a \leftarrow b.$$

$$b \leftarrow .$$

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

If S is a **consistent** *set of literals, then S is an answer set of π iff S^+ is an answer set of π^+.*

In other words $n\_$ can replace $\neg$, but only if the answer set is **consistent**.

Consider the program

$$a \leftarrow b.$$

$$\neg a \leftarrow .$$

$$b \leftarrow .$$

The answer set for the above program is $lit = \{b, \neg a\, a, \neg b\}$. (There is currently a debate in the community that this is not an answer set – that there is no answer set at all for this program.)

Contrast this with the program

$$a \leftarrow b.$$

$$n\_a \leftarrow .$$

$$b \leftarrow .$$

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.

The program

$$\neg a \leftarrow b.$$

$$\neg b \leftarrow .$$

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

$$b \leftarrow a.$$

$$\neg b \leftarrow .$$

has answer set $\{\neg b\}$.

However, these two programs are equivalent in propositional logic. The first program translates to

$$\neg b \Rightarrow \neg a \equiv (b \vee \neg a)$$

and the second program translates to

$$a \Rightarrow b \equiv (\neg a \vee b).$$

In the case of the second program, a is *unknown* with respect to the answer set.

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

An Example

$eligible(X) \leftarrow highGPA(X).$
$eligible(X) \leftarrow special(X), fairGPA(X).$
$\neg eligible(X) \leftarrow \neg special(X), \neg highGPA(X).$
$interview(X) \leftarrow \textbf{not}\ eligible(X), \textbf{not}\ \neg eligible(X).$

$fairGPA(john).$
$\neg highGPA(john).$

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