Beyond Definite Knowledge

CS 475

April 14, 2003

1 The Running Example

Recall that a definite clause KB consists of rules of the form

$$a \leftarrow a_1, \ldots, a_n \tag{1}$$

This type of KB allows us to represent knowledge about various domains, as we have seen so far. Usually, we use rule of this form to represent a clausal knowledge of the form "if $a_1, \ldots, a_n$ are true then a must be true as well". Now, consider the following example.

Example 1.1 We would like to create a student information knowledge base that includes student name, courses taken, grade for each course etc. To make thing simpler, we will assume that the name of each student is unique.

We will use the following predicates (relations) in our representation.

- $student(S)$ – S is a student;
- $course(C)$ – C is a course;
- $enrolled(S, C)$ – student S enrolles in course C;
- $pre\_reg(C, L)$ – L is a list of prerequisite of course C;
- $grade(S, C, G)$ – S got the grade G in the course C.

As we have done previously, our KB will contain information such as

- Atoms about students:

$$student(bob).$$
$$student(tom).$$
$$student(marry).$$
$$student(pat).$$

- Atoms about courses:

$$course(cs475)$$
$$course(math270)$$
$$course(cs482)$$
$$course(cs270)$$

- Atoms about enrollment:

$$enrolled(bob, cs475)$$
$$enrolled(pat, math270)$$
$$enrolled(tom, cs270)$$
$$enrolled(bob, cs482)$$

- Atoms about courses's prerequisite:

$$pre\_reg(cs475, [math270, cs270])$$
$$pre\_reg(cs482, [math270])$$

- Atoms about grade:

$$grade(bob, cs270, a)$$
$$grade(bob, math270, b)$$
$$grade(bob, cs270, a)$$

- Rules representing our knowledge in this domain. For example, the rule saying that " a student can enroll in a course if he finished one possible set of the course prerequisite" is represented as follows.

$$\begin{aligned} passed(S, C) &\leftarrow grade(S, C, G), member(G, [a, b, c]). \\ finished(S, []) &\leftarrow \\ finished(S, [H|T]) &\leftarrow passed(S, H), finished(S, T). \\ can\_enroll(S, C) &\leftarrow student(S), pre\_reg(C, L), finished(S, L). \end{aligned}$$

  The first rule says that student S passed a course C if he/she has the grade a, b, or c in the course. The next two rules state that a student finished a list of course L if he/she passed all the courses in L.

Let us assume that the KB consists of *only* the above rules. We will add more rules to it, whenever it is necessary.

2 Equality and Unique Name Assumption

Consider the query: $?student(X) \leftarrow$.

The answers for this question (I hope that you remember how to get this answers) are $X = bob$, $X = tom$, $X = marry$, or $X = pat$.

In our computation, we make the assumption that $bob \neq tom$, $tom \neq marry$, etc. This is called the **Unique Name Assumption** (UNA), i.e., we assume that terms that are not unifieable represent different objects in our representation. Obviously, the above computation would not be correct if *bob* and *tom* represent the same student.

Before stating the UNA formally, we need some more notation. We say that two terms t_1 and t_2 are *equal*, written as $t_1 = t_2$, if for every interpretation I of the KB, t_1 and t_2 denote the same object. Some of the equalities and rules that we can derive are:

- If X is a variable, $X = X$ – since in every interpretation $I = \langle D, \phi, \pi$ and an variable assignment ρ, X is maped to a constant. So, the left side and the right side represent the same constant. We can represent this as a rule

$$X = X \quad \leftarrow$$

- If $X = Y$ then $Y = X$.

$$Y = X \quad \leftarrow \quad X = Y.$$

- If $X = Y$ and $Y = Z$ then $X = Z$.

$$X = Z \quad \leftarrow \quad X = Y \wedge Y = Z.$$

- For an n-ary function symbol f, we have

$$f(X_1, X_2, \ldots, X_n) = f(Y_1, \ldots, Y_n) \quad \leftarrow \quad X_1 = Y_1 \wedge X_2 = Y_2 \ldots X_n = Y_n.$$

- For an n-ary predicate symbol p, we have

$$p(X_1, X_2, \ldots, X_n) \quad \leftarrow \quad p(Y_1, \ldots, Y_n) \wedge X_1 = Y_1 \wedge X_2 = Y_2 \ldots X_n = Y_n.$$

Formally, the **Unique Name Assumption** (UNA) is as follows:

- $c \neq c'$ for any distinct constants c and c';
- $f(X_1, \ldots, X_n) \neq g(Y_1, \ldots, Y_m)$ for any distinct function symbols f and g;
- $f(X_1, \ldots, X_n) \neq f(Y_1, \ldots, Y_n) \leftarrow X_i \neq Y_i$ for any function symbol f;
- $f(X_1, \ldots, X_n) \neq c$ for any function symbol f and constant c;
- $t \neq X$ for any term t in which X appears and t is not the term X.

To check whether $t_1 \neq t_2$ for two terms t_1 and t_2, we can use the unification algorithm. There are three possible cases:

- t_1 and t_2 are not unifiable, then $t_1 \neq t_2$ holds. For example, because $f(X, a)$ and $f(g(X), a)$ are not unifiable since $X \neq g(X)$ we have that $f(X, a) \neq f(g(X), a)$.
- t_1 and t_2 are identical then $t_1 \neq t_2$ does not hold. For example, for $t_1 = t_2 = X$ we have that $t_1 \neq t_2$ does not hold.
- Otherwise, we can find some interpretation in which $t_1 \neq t_2$ is true and some others in which $t_1 \neq t_2$ is false. For example, for $t_1 = f(W, a, g(Z))$ and $t_2 = f(t(X), X, Y)$, the most general unifier of t_1 and t_2 are $\{X/a, W = t(a), Y = g(Z)\}$, i.e., we can find interpretations in which $t_1 \neq t_2$ does not hold (interpretations in which X, W, Y is assigned a, $t(a)$, $g(Z)$, respectively). However, we have that $t_1 \neq t_2$ is true in interpretations in which the above assigments are not true.

3 Closed World Assumption

Now, consider the query: $?enrolled(bob, cs270) \leftarrow$. It is easy to see that the algorithms we have learned so far will give us the answer *NO*. Why?

We get the answer 'NO' since the algorithm checks all rules from the KB and there exists no rule whose head can unify with $enrolled(bob, cs475)$. Therefore, it stops and gives the answer 'NO'.

This conclusion is true only under the assumption that our KB is *complete*, i.e., we assume that all we know are encoded in the KB. This assumption is called the **Closed World Assumption**. More formally,

The **Closed World Assumption** says that given a knowledge base KB and a grounded atom a, if we can not prove that $KB \models a$ then it holds that $KB \models \neg a$.

The consequence of the CWA: Consider a KB and an atom a. Assume that the rules in KB whose head is a are

$$\begin{array}{lcl} a & \leftarrow & b_1 \\ a & \leftarrow & b_2 \\ & \ldots & \\ a & \leftarrow & b_n \end{array}$$

Then the CWA tells us that $a \Leftrightarrow b_1 \vee \ldots b_n$ where $\Leftrightarrow$ means "if and only if". This is the **Clark's completion** for atom a.

Before we can compute the Clark's completion of a predicate p, we need the definition of **Clark's normal form**. For a clause

$$p(t_1, \ldots, t_n) \leftarrow B$$

the Clark's normal form is the clause

$$p(V_1, \ldots, V_n) \leftarrow V_1 = t_1 \wedge \ldots V_n = t_n \wedge B$$

where V_i's are n new variables which do not appear in the original clause. When the clause is the fact (B is empty) we write $true$ instead of B in the clause's normal form.

The Clark's completion of a predicate p is defined from the set of Clark's normal form for the rules with the predicate p in its head:

$$\begin{array}{lcl} p(V_1, \ldots, V_n) & \leftarrow & B_1 \\ p(V_1, \ldots, V_n) & \leftarrow & B_2 \\ & \ldots & \\ p(V_1, \ldots, V_n) & \leftarrow & B_n \end{array}$$

is

$$p(V_1, \ldots, V_n) \Leftrightarrow B_1 \vee B_2 \ldots \vee B_n.$$

For example, for the predicate $student$ in our KB, the Clark's normal form of the rules whose head is the predicate $student$ are:

$$\begin{array}{lcl} student(X) & \leftarrow & X = bob. \\ student(X) & \leftarrow & X = tom. \\ student(X) & \leftarrow & X = marry. \\ student(X) & \leftarrow & X = pat. \end{array}$$

and hence, the completion of the predicate $student$ is

$$student(X) \Leftrightarrow X = bob \vee X = tom \vee X = marry \vee X = pat.$$

Try to compute what is the completion of the predicate $finished$.

Using the CWA: when we assume the CWA, we can prove that $\neg a$ is true by showing that a cannot be proved. This is called **negation as failure**: we conclude that $\neg a$ is true if we cannot prove a. We will write " not a" to represent the fact that a cannot be proved. With this, we can now allow this type of knowledge

to occur in the body of a rule. For example, we can represent the rule "course C is empty if nobody enrolls in C" by the clauses:

$$\begin{aligned} is\_not\_empty(C) &\leftarrow student(S), enrolled(S,C). \\ empty(C) &\leftarrow course(C) \wedge \text{ not } is\_not\_empty(C). \end{aligned}$$

Note that a seemingly "straightforwad" clause:

$$empty(C) \leftarrow course(C) \wedge \text{ not } enrolled(S,C)$$

will not work due to the free variable S that occurs in the negation as failure atom not $enrolled(S,C)$.

If we allow negation as failure atoms to occur in the body of the rules of the KB, we need to replace not by $\neg$ when computing the Clark's completion.

Example 3.1 For the program

$$\begin{aligned} p &\leftarrow q \wedge \text{ not } r. \\ p &\leftarrow s. \\ q &\leftarrow \text{ not } s. \\ r &\leftarrow \text{ not } t. \\ t &\leftarrow \\ s &\leftarrow w. \end{aligned}$$

The completion of the atoms in this KB are given below:

$$\begin{aligned} p &\Leftrightarrow (q \wedge \neg r) \vee s. \\ q &\Leftrightarrow \neg s. \\ r &\Leftrightarrow \neg t. \\ t &\Leftrightarrow true. \\ s &\Leftrightarrow w. \\ w &\Leftrightarrow false. \end{aligned}$$

Now that we add negation as failure atoms to the KB, we need to change the proof procedures for answering queries that we have learned previously to accomodate this new type of atoms.

Bottom-Up Proof Procedure

$C := \emptyset$
repeat
  either **select clause** $h \leftarrow b_1 \wedge \ldots \wedge b_m$ in KB such that
    $b_i \in C$ for all i, and $h \notin C$
  $C := C \cup \{h\}$

  or **select clause** $h \leftarrow b_1 \wedge \ldots \wedge b_m$ in KB such that
    either for some $b_i \in C$, not $b_i \in C$
    or for some $b_i =$ not g, $g \in C$
  $C := C \cup \{$ not $h\}$

until no more clauses can be selected.

Example 3.2 For the program KB_0

$$
\begin{array}{lll}
p & \leftarrow & q \wedge \text{ not } r. \\
p & \leftarrow & s. \\
q & \leftarrow & \text{not } s. \\
r & \leftarrow & \text{not } t. \\
t & & \\
s & \leftarrow w. &
\end{array}
$$

the atom added to C in the sequence is t, not r, not w, not s, q, p. This allows us to conclude, for example, that p and q are true ($KB_0 \models p$, $KB_0 \models q$); w is false ($KB_0 \models \neg w$);

Top-Down Proof Procedure: Conclude not a if for every rule

$$a \leftarrow b_i$$

the body b_i fails after finitely many steps. In the derivation tree, whenever we get a *not*a as a goal, we need to try to prove a and if the proof fails, we conclude not a and then continue where we left off.

Example 3.3 For the program KB_0 (same as in the previous example)

$$
\begin{array}{lll}
p & \leftarrow & q \wedge \text{ not } r. \\
p & \leftarrow & s. \\
q & \leftarrow & \text{not } s. \\
r & \leftarrow & \text{not } t. \\
t & & \\
s & \leftarrow w. &
\end{array}
$$

and the query $?p$, the derivation is:

- $yes \leftarrow p$. (select the first rule)
- $yes \leftarrow q \wedge$ not r. (select the first atom: q)
- $yes \leftarrow$ not $s \wedge$ not r. (select the first atom: not s)
- Need to prove not s means that we need to show that s cannot be proved.
  - $yes \leftarrow s$. (select the last rule)
  - $yes \leftarrow w$. fails because no rule with the head w available.

  The above derivation shows that not s is true. So, we need to show now that not r is true. Again, this means that we need to show that $?r$ is false.
  - $yes \leftarrow r$. (select the last rule)
  - $yes \leftarrow$ not t. Means that we need to check for t. This is true, and hence not t fails. Therefore, the proof for r fails. Hence, not r is true.
- Both negation as failure atoms in the body of the answer are true, so we can conclude that p is true.

Problem with variables: The top-down proof procedure is only correct when the derivation does not *flounder*: a situation when free variables in the negation as failure atoms never get a binding to a constant. Delaying the negation as failure goal can be helpful.

Example 3.4

$$\begin{array}{rcl} p(X) & \leftarrow & \text{not } q(X) \wedge r(X). \\ q(a) & \leftarrow & \\ q(b) & \leftarrow & \\ r(d) & & \end{array}$$

and the query $?p(X)$ will get into problem if we want to prove not $q(X)$ first.

If we select $r(X)$ first, we get the answer $X = d$.

Sometime, delaying negation as failure goal will not help:

Example 3.5

$$\begin{array}{rcl} p(X) & \leftarrow & \text{not } q(X). \\ q(X) & \leftarrow & \text{not } r(X). \\ r(a) & & \end{array}$$

and the query $?p(X)$, we cannot delay the goal not $q(X)$.

4 Disjunctive and Negative Knowledge

Let us consider again our KB about student. We would like to add the information to the KB: "each course is taught by an instructor". We can see that this can be easily added by introducing the predicate $taught\_by(C, I)$ which says that instructor I teaches course C. Consider the following situation:

- We want to add the following knowledge to the KB:

  The course *cs270* is taught by Enrico or Hing.

  We do not know how to express this type of knowledge. This type of knowledge is called *disjunctive knowledge*. In order to represent this type of information, we need to introduce *disjunction* in the head of the clauses. The above knowledge can be represented by the rule:

  $$taught\_by(cs270, enrico) \vee taught\_by(cs270, hing) \leftarrow \quad (2)$$

- We want to add the following knowledge to the KB:

  If a student missed more than 6 classes then he cannot get the grade A. (this is fictional:-)

  Again, we do not know how to express this type of knowledge. We can add a predicate like $limit(C, N)$ to say that the course C has a limit of N missing days and $absent(S, C, N)$ to say that S missed N classes in cource C but we cannot express the fact that if a student misses more than N days, he cannot get the grade A. This type of knowledge is called *negative knowledge*. In order to represent this type of information, we need to introduce *negation* in the head of the clauses. The above knowledge can be represented by the rule:

  $$\neg grade(S, C, a) \leftarrow limit(C, L), absent(S, C, N), N > L. \quad (3)$$

We extend the language of definite clauses (remember: *variables, terms, atoms, clauses* and the *not*) with the following:

- A *literal* is either an atom a or its negation $\neg a$.
- A *general clause* is of the form

$$L_1 \vee \ldots \vee L_k \leftarrow B_1 \wedge \ldots \wedge B_m \wedge \text{ not } B_{m+1} \wedge \ldots \wedge \text{ not } B_{m+k} \quad (4)$$

  where the L_i's and B_j's are **literals**. Under this definition, the rules that we wrote above are general clauses.

A disjunctive program (or a KB) is a set of general clauses. Intuitively, the rule (4) says that whenever the B_j's are true at least one of the L_i's is true. For example, the rule (2) says that at least one of the atom $taught\_by(cs270, enrico)$ or $taught\_by(cs270, hing)$ must be true.

Now that we add extra syntactical elements to the KB we also need to change the semantics, i.e., we need to specify what does a program entail? There are several interesting questions related to this problems which give raise to different semantics of disjunctive programs. We will study one of them: *the anwer set semantics*. But first, the problems:

- **No model**: Consider the story "In the town of Casablanca, the barber named *Tom* shaves everyone who does not shave themselve". Intuitively, this can be represent by a program, called P_1, that consists of one single general clause:

$$shaves(tom, Y) \leftarrow \text{ not } shaves(Y, Y).$$

  Now, we can show that for every individual $p \neq tom$ from the town, who does not shave himself, *tom* shaves p, i.e., $shaves(tom, p)$ is true. What about *tom*? Does he shave himself? Who shaves him? This program does not have a model.

- **Too many models:** Consider the story "The old traffic light has only two colors: red and green. Whenever the red light is not o, the green light is on, and vice versa." This piece of information can be represented by the program, called P_2, as follows.

$$\begin{aligned} red &\leftarrow \text{ not } green. \\ green &\leftarrow \text{ not } red. \end{aligned}$$

  Intuitively, this program does have two models: $\{red, \text{ not } green\}$ and $\{green, \text{ not } red\}$.

The above discussion shows that in presence of disjunctive and negated knowledge, a program might not have a model or have more than one models. Is it a problem? What is a model of a general program? Studying this question has been the subject of intense research in logic programming and database for several years. We will pay our attention to the *answer set* definition by Gelfond and Lifschitz.

5 Answer Sets of Disjunctive Programs

We will define the notion of an *answer set* of a program. Since we can always replace a rule with variables by the set of its ground instantiations, we will assume that programs in this section do not contain variables. Let us first recall the rule of a disjunctive logic program:

$$l_1 \vee \ldots \vee l_k \leftarrow b_1 \wedge \ldots \wedge b_m \text{ not } b_{m+1} \wedge \ldots \wedge \text{ not } b_{m+k} \quad (5)$$

The definition is defined in two steps. First, we will deal with programs that do not contain *not* ($k = 0$ in every rule of P). Next, we will do with programs that contain *not*.

Definition 5.1 *Let P be a program without not. A set of literals S is an answer set of P if S is a* minimal set *of literals satisfying the following conditions:*

- *For every rule*
$$l_1 \vee \ldots \vee l_n \leftarrow b_1 \wedge \ldots \wedge b_m$$
of P, if $\{b_1, \ldots, b_m\} \subseteq S$, then at least one of the l_is' belongs to S, i.e., $\{l_1, \ldots, l_n\} \cap S \neq \emptyset$.
- *If there exists an atom a such that $a \in S$ and $\neg a \in S$ then S is the set of all literals of the program P.*

Let P now be a general program (it might contain *not*). For a set of literals X, by P^X we denote the program obtained from P by

- Deleting any rule $l_1 \vee \ldots \vee l_n \leftarrow b_1 \wedge \ldots \wedge b_m$ not b_{m+1}, not b_{m+k} for that $\{b_{m+1}, \ldots, b_{m+k}\} \cap X \neq \emptyset$, i.e., the body of the rule contains a negated atom not b_l and b_l belongs to X; and
- Removing all of the negated atoms from the remaining rules.

Note: P^X is a program without *not*.

Definition 5.2 *A set of atoms X is called a* answer set *of a program P if X is an answer set of the program P^X.*

In the next examples, we illustrate the above definition.

Example 5.1 The program
$$taught\_by(cs270, enrico) \vee taught\_by(cs270, hing) \leftarrow$$
has two answer sets: $S_1 = \{taught\_by(cs270, enrico)\}$ and $S_2 = \{taught\_by(cs270, hing)\}$.

Example 5.2 The program P
$$shaves(tom, Y) \leftarrow \text{not } shaves(Y, Y).$$
does not have an answer set. Here is why it is so. First, we obtained a program without variables by grounding the rules (We learned how to do this in the second and third chapter). Because we have only one constant *tom*, we obtain the following program, called Q:
$$shaves(tom, tom) \leftarrow \text{not } shaves(tom, tom).$$
The set of literals of this program consists of two literals: $\{shaves(tom, tom), \neg shaves(tom, tom)\}$. Therefore, there are four possibilities: $S_1 = \emptyset$, $S_2 = \{shaves(tom, tom)\}$, $S_3 = \{\neg shaves(tom, tom)\}$, and $S_4 = \{shaves(tom, tom), \neg shaves(tom, tom)\}$. We have that

- Q^{S_1} is the program consisting of the rule: $shaves(tom, tom)$. which has only one answer set $A = \{shaves(tom, tom)\}$. Since $A \neq S_1$, S_1 is not an answer set of Q.
- Q^{S_2} is the empty program which has only one answer set $A = \emptyset$. Since $A \neq S_2$, S_2 is not an answer set of Q.
- Q^{S_3} is the program consisting of the rule: $shaves(tom, tom)$. which has only one answer set $A = \{shaves(tom, tom)\}$. Since $A \neq S_3$, S_3 is not an answer set of Q.

- Q^{S_4} is the empty program which has only one answer set $A = \emptyset$. Since $A \neq S_4$, S_4 is not an answer set of Q.

This shows that the program Q (and hence, P) does not have any answer set.

Example 5.3 The program P with two rules:

$$\begin{array}{rcl} red & \leftarrow & \text{not } green. \\ green & \leftarrow & \text{not } red. \end{array}$$

has two answer sets: $S_1 = \{red\}$ and $S_2 = \{green\}$ because

- P^{S_1} is a program consisting of the rule "red" that has a unique answer set $\{red\}$, which equals S_1. Hence, S_1 is an answer set of P.
- P^{S_2} is a program consisting of the rule "$green$" that has a unique answer set $\{green\}$, which equals S_2. Hence, S_2 is an answer set of P.

We can easily prove that any set of literals that is different than S_1 or S_2 cannot be an answer set of P.

The above examples show that if we want to find an answer set of a program, we can follow the generate-and-test algorithm. That is, we generate a set of literals of the program, then check whether it is an answer set or not. This is certainly not a good method to compute an answer set since the number possible answer sets grows exponential on the number of literals of a program. To reduce the number of guesses, we can use the following observations:

- If p is a fact of the program P, then every answer set of p must contain p;
- If p does not occur in the head of any rule of the program P, then p does not belong to any of the answer set of P;
- If X is an answer set of P, $l_1, \ldots, l_n$ belongs to X, $l_{n+1}, \ldots, l_{n+m}$ do not belong to X, and

  $$l_0 \leftarrow l_1, \ldots, l_n, \text{ not } l_{n+1}, \ldots, \text{ not } l_{n+m}$$

  is a ground rule of P then l_0 must belong to X.

The next example demonstrates the usefulnes of these observations.

Example 5.4 Let the program P be

$$\begin{array}{rcl} p & \leftarrow & \\ \neg q & \leftarrow & p \\ r & \leftarrow & \neg p \\ t & \leftarrow & \neg q, \text{ not } s \\ u & \leftarrow & \text{not } \neg u \end{array}$$

Let $lit(P)$ be the set of literals of P. We have that $lit(P) = \{p, q, r, t, u, s, \neg p, \neg q, \neg r, \neg t, \neg s, \neg u\}$. There are 2^{12} possible subsets of $lit(P)$. It is too difficult to check one-by-one whether a subset of $lit(P)$ is an answer set of P. Observer that

- p is a fact of the program P. So, every answer set X of P must contain p. (observation 1)
- Since p belongs to every answer set and
$$\neg q \leftarrow p$$
is a ground rules of P, $\neg q$ must belong to every answer set of P.
- Because there is no rule whose head is s, and $\neg q$ belongs to every answer set of P, from the rule $t \leftarrow \neg q,\ \ \text{not}\ \ s$, we can conclude that t belongs to every answer set of P.
- Since there is no rule with $\neg u$ in the head, we have that $\neg u$ does not belong to any answer set of P; This leads to the conclusion that u belongs to every answer set of P.

The about observation show that $X = \{p, \neg q, t, u\}$ is part of every answer set of P. That is, every answer set A of P equals $X \cup Y$ where Y is a possible subset of $\{r, s, \neg r, \neg s\}$. That is, there are at most 16 answer sets. This shows how we can reduce the number of guesses in computing the answer sets.

Now, we will show that X is indeed an answer set of P. We have that P^X consists of the following rules:

$$\begin{array}{rcl} p & \leftarrow & \\ \neg q & \leftarrow & p \\ r & \leftarrow & \neg p \\ t & \leftarrow & \neg q \\ u & \leftarrow & \end{array}$$

X satisfies all the rules of P^X and it is minimal (removing any element from X, we obtain a set that does not satisfy all the rules of P^X). So, X is an answer set of P^X, and hence, an answer set of P.

We have proved that X is an answer set of P. We also proved that any answer set of P would contain X. This implies that X is the unique answer set of P.

6 Answer set semantics

In the next definition, we discuss the answers to the query "?l", given a program P and a literal l.

Definition 6.1 *Let P be a program and l be a literal. We say that*

- *P entails l (l is* true*, given P), denoted by $P \models l$, if l belongs to every answer set of P.*
- *P does not entails l (l is* false*, given P), if l does not belong to any answer set of P.*
- *P might entail l (l is* unknown*, given P), if it is not true, nor false given P (i.e., l belongs to some answer sets and does not belong to some (other) answer sets of P).*

The difference between the above definition (also called the **answer set semantics**) and what we have learned so far is that we may answer a query with 'unknnown'. Consider again the example, the program P consisting of a single rule

$$taught\_by(cs270, enrico) \vee taught\_by(cs270, hing) \leftarrow$$

we know that this program has two answer sets $S_1 = \{taught\_by(cs270, enrico)\}$ or $S_2 = \{taught\_by(cs270, hing)\}$. Consider the different queries:

- $P \models taught\_by(cs270, enrico)$? The answer to this query is 'unknown' since $taught\_by(cs270, enrico)$ belongs to one answer set (S_1) and does not belong to the other one (S_2). *This is something lile "Enrico might or might not teach cs270".*
- $P \models \neg taught\_by(cs270, enrico)$? The answer to this query is 'false' since $\neg taught\_by(cs270, enrico)$ does not belong to any answer set. *This answer expresses that "it is definitely false to say that Enrico does not teach cs270."*

7 Summarize

The following definitions/topics are discussed in this chapter:

- The unique name assumption
- The closed world assumption
- Clark's completion of atoms and predicates
- Negation as failure
- Logic programs with negation as failure
- Disjunctive and negative knowledge: disjunctive logic programs
- Answer sets of disjunctive logic programs, entailment based on answer sets