Lecture 3: Normal Programs

Outline
1. Negative conditions
2. Stable model semantics
3. Variables and domains
4. Programming tips
5. Problem solving

1. NEGATIVE CONDITIONS
➤ The semantics based on least models provides a logical foundation for rule-based reasoning: \( P \models a \) iff \( a \in \text{LM}(P) \) for an atom \( a \).
➤ In particular, atoms \( a \in \text{Hb}(P) \) that are not logical consequences of \( P \), i.e., \( P \notmodels a \) holds, are false in \( \text{LM}(P) \) by default.
➤ In many applications, it is convenient/necessary to refer to complements of certain relations using negative conditions.
➤ The notion of answer sets based on stable models provides a declarative semantics for programs involving negative conditions.

Example. Consider the following definition of a conscript:

\[
\text{Conscript}(X) \leftarrow \text{Person}(X), \neg \text{Female}(X).
\]

2. STABLE MODEL SEMANTICS
➤ In 1988, Gelfond and Lifschitz proposed stable models in order to provide a declarative semantics for negative conditions in rules.
➤ The rules of normal logic programs are of the form

\[
a \leftarrow b_1, \ldots, b_n, \neg c_1, \ldots, \neg c_m.
\]

where \( \neg \) denotes negation by default.
➤ Stable models are based on the following two ideas:
1. \( M \models \neg c \) holds for a negative condition \( \neg c \iff c \notin M \), and
2. a model \( M \) is stable iff it is the least Herbrand model for the rules having their all negative conditions satisfied by \( M \).

Example
Consider the following set of rules involving negative conditions.

\[
\begin{align*}
\text{Conscript}(x) & \leftarrow \text{Person}(x), \neg \text{Female}(x). \\
\text{Female}(x) & \leftarrow \text{Person}(x), \neg \text{Volunteer}(x), \neg \text{Conscript}(x). \\
\text{Person}(joe) & \leftarrow.
\end{align*}
\]

What would be the right answer for the query \( \text{Conscript}(joe) \)?
➤ The meaning of the rules depends on the order of application:

\[
\begin{align*}
\text{Person}(joe), \neg \text{Female}(joe) & \implies \text{Conscript}(joe) \\
\text{Person}(joe), \neg \text{Volunteer}(joe), \neg \text{Conscript}(joe) & \implies \text{Female}(joe) \\
\end{align*}
\]
➤ Thus it seems non-trivial to combine recursive definitions with negation and, in particular, to obtain a declarative semantics.
Example

Reconsider the program from the preceding example after grounding:

Conscript(joe) ← Person(joe), ~Female(joe).
Female(joe) ← Person(joe), ~Volunteer(joe), ~Conscript(joe).
Person(joe).

The model $M = \{\text{Person}(joe), \text{Conscript}(joe)\}$ is stable.

The negative conditions of the first and the last rule are true in $M$ which is the least Herbrand model of the respective positive rules:

Conscript(joe) ← Person(joe).  Person(joe).

But $N = \{\text{Person}(joe), \text{Female}(joe)\}$ is also stable (which suggests us to specify Joe's gender; or to revise the given rules somehow).

© 2007 TKK / TCS

Definition of Stability

Definition. Let $P$ be a normal logic program without variables and $M \subseteq \text{Hb}(P)$ an interpretation.

The Gelfond-Lifschitz reduct of $P$ with respect to $M$ is

$$P^M = \{a \leftarrow b_1, \ldots, b_n \mid a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m \in P$$

and $M \models \sim c_1, \ldots, \sim c_m\}.$$

Remark. Note that in the definition of $P^M$,

$M \models \sim c_1, \ldots, \sim c_m$ iff $M \cap \{c_1, \ldots, c_m\} = \emptyset$.

Definition. Let $P$ be a normal logic program without variables.

An interpretation $M \subseteq \text{Hb}(P)$ is a stable model of $P$ iff $M = \text{LM}(P^M)$.

© 2007 TKK / TCS

Example

Consider a normal logic program $P$ having the rules listed below:

$$a \leftarrow c, \sim b.$$

$$b \leftarrow \sim a.$$

$$c \leftarrow \sim d.$$

$$d \leftarrow \sim a.$$

1. The interpretation $M_1 = \{a, c\}$ is a stable model of $P$ because $P^{M_1} = \{a \leftarrow c. \ c. \}$ and $M_1$ is the least model of $P^{M_1}$.

2. But $M_2 = \{a, d\}$ is not stable because $P^{M_2} = \{a \leftarrow c. \}$ for which the least model is $\emptyset$. Note that $M_2 \models P$ in the classical sense.

3. Finally, $M_3 = \{b, d\}$ is also a stable model of $P$.

© 2007 TKK / TCS

The $\Gamma_P$ Operator

Definition. Given a normal logic program $P$, define an operator $\Gamma_P : 2^{\text{Hb}(P)} \to 2^{\text{Hb}(P)}$ by setting

$$\Gamma_P(M) = \{a \mid a \in \text{Hb}(P) \text{ and } P^M \models a\} = \text{LM}(P^M).$$

Proposition. An interpretation $M \subseteq \text{Hb}(P)$ is a stable model of a normal program $P$ iff $M = \Gamma_P(M)$.

The operator $\Gamma_P$ is not monotonic but antimonotonic.

Proposition. For any normal program $P$ and interpretations $M \subseteq N \subseteq \text{Hb}(P)$, $\Gamma_P(N) \subseteq \Gamma_P(M)$.

Proof. It is sufficient to note that $M \subseteq N$ implies $P^N \subseteq P^M$ and $\text{LM}(P^N) \subseteq \text{LM}(P^M)$ by the monotonicity of $\text{LM}(\cdot)$.  \qed
Properties of Stable Models

- Unlike the least model of a positive program, stable models are not necessarily unique as demonstrated by programs given below:
  1. $P_0 = \{ a \leftarrow \neg a. \}$ has no stable models.
  2. $P_1 = \{ a \leftarrow \neg b. \}$ has one stable model $\{ a \}$.
  3. $P_2 = \{ a \leftarrow \neg b. b \leftarrow \neg a. \}$ has two stable models $\{ a \}$ and $\{ b \}$.

- Stable models are minimal in the sense that if $M \in \text{SM}(P)$ then there is no other $N \in \text{SM}(P)$ such that $N \subseteq M$.
- A stable model $M \in \text{SM}(P)$ is strongly grounded in the rules of $P$: $a \in M$ iff $P^M \models a$.

3. VARIABLES AND DOMAINS

The ground program $\text{Gnd}(P)$ is defined for normal logic programs $P$ in the same way as for positive programs.

**Definition.** Let $P$ be a normal logic program—potentially involving variables—and $\text{Gnd}(P)$ the respective ground program.

A Herbrand interpretation $M \subseteq \text{Hb}(P)$ is a stable model of $P$ iff $M = \Gamma_{\text{Gnd}(P)}(M) = \text{LM}(\text{Gnd}(P)^M)$.

**Example.** Let us consider $P = \{ A(c, d). \ B(x) \leftarrow A(x, y), \neg B(y). \}$. The ground program $\text{Gnd}(P)$ contains the following rules:

- $A(c, d)$.
- $B(c) \leftarrow A(c, c), \neg B(c)$.
- $B(c) \leftarrow A(c, d), \neg B(d)$.
- $B(d) \leftarrow A(d, c), \neg B(c)$.
- $B(d) \leftarrow A(d, d), \neg B(d)$.

The interpretation $M = \{ A(c, d), B(c) \}$ is the only stable model of $P$.

Answer Set Programming

- A traditional PROLOG system answers a query $Q$ either “yes” (with an answer substitution $\theta$ for the variables of $Q$) or “no”.

- Stable models, or answer sets, are based on a novel interpretation of logic programs as sets of constraints on their models.

- Typically, an answer set—computed using a special search engine—captures a solution to the problem being solved.

- Rule-based languages are highly expressive: Many problems involving constraints can be reformulated as problems of finding a stable model for the respective set of rules.

Domain Predicates

- Ground programs $\text{Gnd}(P)$ can become very large and they may contain many useless or redundant rules.

- A way to prune unnecessary rules is to introduce domain predicates which are relation symbols having a fixed interpretation.

- Even recursive definitions for domain predicates, like $G(\cdot, \cdot)$ below, can be tolerated unless recursion does not involve negation.

**Example.** Consider the following example:

- $D(a)$.
- $E(b)$.
- $F(x) \leftarrow D(x)$.
- $F(x) \leftarrow E(x)$.
- $G(x, y) \leftarrow D(x), E(y)$.
- $G(y, x) \leftarrow G(x, y), F(x), F(y)$.
- $R(x, y) \leftarrow G(x, y), \neg S(y, x)$.
- $S(y, x) \leftarrow G(x, y), \neg R(y, x)$.

Here $D$, $E$, $F$, and $G$ are domain predicates but $R$ and $S$ are not.
Some observations about the preceding program, say $P$, follow:

- The Herbrand universe $\text{Hu}(P) = \{a, b\}$ is finite.
- The least Herbrand model for $P'$ consisting of the first six rules of $P$ is $\text{LM}(\text{Gnd}(P')) = \{D(a), E(b), F(a), F(b), G(a, b), G(b, a)\}$.
- The model $\text{LM}(\text{Gnd}(P'))$ can be represented as a set of facts.
- Only two ground instances of the last two rules each are needed:
  - $R(b, a) \leftarrow G(a, b), S(b, a)$.
  - $R(a, b) \leftarrow G(b, a), S(a, b)$.
- $S(b, a) \leftarrow G(a, b), \sim S(b, a)$.
- $S(a, b) \leftarrow G(b, a), \sim R(a, b)$.
- An intelligent grounder can simplify these rules further by dropping conditions $G(a, b)$ and $G(b, a)$ as they are satisfied for sure.

### 4. PROGRAMMING TIPS

The logical connectives of propositional logic are available.

- The conjunction of conditions $c_1, \ldots, c_n$ is captured by a single (positive) rule $c \leftarrow c_1, \ldots, c_n$.
- Expressing the disjunction of conditions $d_1, \ldots, d_n$ requires the introduction of $n$ rules $d \leftarrow d_1, \ldots, d_n$.
- A constraint $\leftarrow b_1, \ldots, b_n$ that formalizes the negation $\neg(b_1 \land \ldots \land b_n)$ is best expressed using a rule $f \leftarrow b_1, \ldots, b_n, \sim f$ where $f$ is a new atom not appearing elsewhere in the program.

**Example.** One is supposed to have one or two delicacies out of three: Some $\leftarrow$ Cake. Some $\leftarrow$ Bun. Some $\leftarrow$ Cookie. All $\leftarrow$ Cake, Bun, Cookie. $F \leftarrow$ All, $\sim F$. $F \leftarrow \sim$Some, $\sim F$.

### Restricting Domains of Variables

- The idea is to control the size of the resulting ground program by introducing domain predicates that fix the domain of each variable.

**Definition.** A normal program $P$ is **strongly typed** or **strongly domain restricted** iff for each rule

$$ R(\overline{t}) \leftarrow R_1(\overline{u_1}), \ldots, R_n(\overline{u_n}), \sim S_1(\overline{u_1}), \ldots, \sim S_m(\overline{u_m}) $$

of $P$ and for each variable $x$ appearing in the rule, $x$ appears in some of the positive conditions $R_i(\overline{u_i})$ where $R_i$ is a domain predicate.

**Example.** Assuming that $D(\cdot)$ is the only domain predicate, the rule

$$ R(x, y) \leftarrow D(x), D(y), \sim S(y, x) $$

is strongly typed, but the rules $F(x, y) \leftarrow D(x), E(x)$ and $E(x) \leftarrow \sim D(x)$ are not.

### Making Choices

- A choice between two atoms $a$ and $b$ can be expressed in terms of two normal rules $a \leftarrow \sim b$ and $b \leftarrow \sim a$.
- Such a choice can be generalized for any number of atoms and conditionalized by adding conditions in rule bodies.
- A typical approach in ASP is to express a number of choices and then exclude certain combinations using other rules or constraints.

**Example.** One is supposed to have coffee or tea—but not both—and also one of three delicacies in case tea is selected:

- Coffee $\leftarrow \sim$Tea. Cake $\leftarrow$ Tea, $\sim$Cookie, $\sim$Bun.
- Tea $\leftarrow \sim$Coffee. Bun $\leftarrow$ Tea, $\sim$Cookie, $\sim$Cake.
- Cookie $\leftarrow$ Tea, $\sim$Bun, $\sim$Cake.
Rules with Exceptions

- Normal programs enable context-dependent reasoning in which the applicability of rules depends dynamically on the context.
- In common-sense reasoning, it is typical to formalize the normal state of affairs including any exceptions to that.

Example. Birds do normally fly—unless we have an exceptional bird.

Flies(x) ← Bird(x), ~Abnormal(x).
Abnormal(x) ← Penguin(x). Abnormal(x) ← Oily(x). . .

The stable models of this program, say P, behave as follows:
1. SM(P ∪ {Bird(tw). }) = {{Bird(tw), Flies(tw)}}.
2. SM(P ∪ {Bird(tw). Oily(tw). }) = {{Bird(tw), Oily(tw), Abnormal(tw)}}.

Example

Consider the translation of \( S = \{ a \lor b, a \lor \neg b, \neg a \lor \neg b \} \) into a normal program. The translation \( P_S \) consists of the following rules:

\[
\begin{align*}
a & \leftarrow \neg \bar{a}, \\
\bar{a} & \leftarrow \neg a, \\
b & \leftarrow \neg \bar{b}, \\
\bar{b} & \leftarrow \neg b.
\end{align*}
\]

A number of observations can be made:
- Now, the set of clauses \( S \) has a model \( M \) iff the program \( P_S \) has a stable model \( N \) such that \( M = N \cap \{a,b\} \).
- Because \( N_1 = \{a,\bar{b}\} \) is a stable model of \( P_S \), we know that \( M_1 = \{a\} \) is a model of \( S \).
- On the other hand, \( N_2 = \{\bar{a},\bar{b}\} \) is not a stable model of \( P_S \).

Graph 3-Coloring

A graph \( G \) can be represented by facts of the form “Edge(x,y),” where \( x \) and \( y \) stand for nodes. The following normal program \( P_G^{bc} \) is a uniform encoding for the problem of coloring the nodes of \( G \) with three colors so that the endpoints of each edge have different colors.

\[
\begin{align*}
\text{Node}(x) & \leftarrow \text{Edge}(x,y). \quad \text{Node}(y) \leftarrow \text{Edge}(x,y). \quad \text{(projection)} \\
\text{Black}(x) & \leftarrow \text{Node}(x), \neg \text{White}(x), \neg \text{Grey}(x). \quad \text{(choices)} \\
\text{White}(x) & \leftarrow \text{Node}(x), \neg \text{Black}(x), \neg \text{Grey}(x). \\
\text{Grey}(x) & \leftarrow \text{Node}(x), \neg \text{White}(x), \neg \text{Black}(x). \\
\text{F} & \leftarrow \text{Edge}(x,y), \text{Black}(x), \text{Black}(y), \neg \text{F}. \quad \text{(constraints)} \\
\text{F} & \leftarrow \text{Edge}(x,y), \text{White}(x), \text{White}(y), \neg \text{F}. \\
\text{F} & \leftarrow \text{Edge}(x,y), \text{Grey}(x), \text{Grey}(y), \neg \text{F}.
\end{align*}
\]

Proposition. The graph \( G \) has a 3-coloring iff \( P_G^{bc} \) has a stable model.
Hamiltonian Cycles in Graphs

The problem is to check whether a given graph has a Hamiltonian cycle which visits all nodes of the graph exactly once. In addition to the edge relation, the following rules are introduced in program $P_{HC}$.

1. The nodes of the graph are extracted from the edge relation:
   \[ \text{Node}(x) \leftarrow \text{Edge}(x,y). \quad \text{Node}(y) \leftarrow \text{Edge}(x,y). \quad \text{Same}(x,x) \leftarrow \text{Node}(x). \]

2. Any cycle starts from a particular node chosen here.
   \[
   \begin{align*}
   \text{Start}(x) \leftarrow \text{Node}(x), \sim \text{Other}(x). \\
   \text{Other}(x) \leftarrow \text{Node}(x), \sim \text{Start}(x). \\
   F \leftarrow \text{Start}(x), \text{Start}(y), \sim \text{Same}(x,y), \text{Node}(x), \text{Node}(y), \sim F. \\
   \text{HasStart} \leftarrow \text{Start}(x), \text{Node}(x). \\
   F \leftarrow \sim \text{HasStart}, \sim F.
   \end{align*}
   \]

3. Next the edges which are on the cycle are chosen.
   \[
   \begin{align*}
   \text{In}(x_1,x_2) & \leftarrow \text{Edge}(x_1,x_2), \sim \text{Out}(x_1,x_2). \\
   \text{Out}(x_1,x_3) & \leftarrow \text{In}(x_1,x_2), \sim \text{Same}(x_2,x_3), \text{Edge}(x_1,x_2), \text{Edge}(x_1,x_3). \\
   \text{Out}(x_3,x_2) & \leftarrow \text{In}(x_1,x_2), \sim \text{Same}(x_2,x_3), \text{Edge}(x_1,x_2), \text{Edge}(x_3,x_2).
   \end{align*}
   \]

4. All nodes of the graph must be reachable via the cycle.
   \[
   \begin{align*}
   \text{Reached}(x) & \leftarrow \text{Start}(x). \\
   \text{Reached}(x) & \leftarrow \text{In}(y,x), \text{Reached}(y), \text{Edge}(y,x). \\
   F & \leftarrow \text{Node}(x), \sim \text{Reached}(x), \sim F.
   \end{align*}
   \]

**Proposition.** The program $P_{HC}$—together with facts that describe the edge relation—has a stable model $\iff G$ has a Hamiltonian cycle.