A Basic Representation and Reasoning System

CMPT 411/721

Reasoning with Definite Clauses

- We next define a simple KR system based on *definite clauses*.
- A definite clause can be thought of as a simple rule, with no negation in the head or body of the rule.
- This language is quite restricted, but we can still define entailment and inference, etc.
- In general, a KB will consist of *facts* and *rules*, and we will be interested in deriving other facts.

The Definite Clause Language: Vocabulary

- Assume that an agent's knowledge is made up of two components:
  - A database of *facts* about the domain (or *ground atomic formulas*)
    E.g. $Mother(jane, paul)$, $Male(arvind)$.
  - A collection of *rules* (or *definite clauses*)
    E.g.
    $Parent(X, Y) \Leftarrow Mother(X, Y)$
    $Gf(X, Y) \Leftarrow Father(X, Z) \wedge Parent(Z, Y)$
- Note that implication is written in the reverse direction from normal.
- Variables are implicitly universally quantified.
- Variables are local to a clause.

The Definite Clause Language: Vocabulary

The vocabulary of our language is made up of:

1. Logical symbols: "(", ")", ",", "$\Leftarrow$", "$\wedge$", "."
    - Note that $\neg$ and $\vee$ aren't included.
2. Non-logical symbols:
    - Constants, predicate symbols, function symbols
        - Uncapitalised strings.
        - Meaning of a string is implicit in its use.
        - E.g.: *johnQsmith*, *bestFriendOf*.
    - Variables
        - Written as capitalised strings.
        - E.g.: X, X_1, *Variable*.

The Definite Clause Language: Syntax

As in FOL, the language expresses

- *terms* that denote objects in the domain and
- *formulas* that make assertions about the domain.

The Definite Clause Language: Terms

A *term* is either

- a variable,
- a constant, or
- an expression of the form $f(t_1, \ldots, t_n)$ where f is a function symbol, and each t_i is a term.

The Definite Clause Language: Formulas

- Formulas are defined as follows:
  - An *atomic formula (atom)* is of the form p or $p(t_1, \ldots, t_n)$ where p is a predicate symbol, and each t_i is a term.
  - A *body* is of the form $a_1 \wedge \cdots \wedge a_n$ where each a_i is an atom.
  - A *definite clause* is of the form

    $a.$ or $a \Leftarrow b$

    where a, the *head*, is an atom and b is a body.
- A *knowledge base* is a set of definite clauses.
- Although it isn't part of the language, a *query* is conventionally written in the form $?b.$ where b is a body.

Example

Example

- (Ground) atomic formulas:

  $$father(ian, sue)$$
  $$father(fred, chris)$$
  $$mother(michelle, chris)$$
  $$num(0)$$

- Definite clauses:
  $\langle$*the above atomic formulas*$\rangle$

  $$gf(ian, chris) \Leftarrow father(ian, fred) \wedge father(fred, chris)$$
  $$gf(X, Y) \Leftarrow father(X, Z) \wedge father(Z, Y)$$
  $$num(s(N)) \Leftarrow num(N)$$
  $$num(X) \Leftarrow father(X, Y)$$

Semantics

- Meaning is attaced to symbols the same as in FOL.
- An interpretation is a pair $\mathcal{I} = \langle D, I \rangle$ where
  1. $D \neq \emptyset$ is the domain .
  2. I is a mapping that assigns
     - to each constant: an element of D
     - to each n-ary function symbol: a mapping from $D^n \Rightarrow D$ and
     - to each n-ary predicate symbol: a subset of D^n
       (0-ary predicate symbols are assigned *true* or *false* in an interpretation.)

Semantics (continued)

- We first give a semantics for variable-free or *ground* expressions:
  - Each ground term denotes an individual in the domain:
    - Constant c denotes the individual $I(c)$ in $\mathcal{I}$.
    - $f(t_1, \ldots, t_n)$ denotes the individual $I(f)(t'_1, \ldots, t'_n)$ in $\mathcal{I}$, where t'_i is the individual denoted by t_i (i.e. $t'_i = I(t_1)$).

Semantics (continued)

- We first give a semantics for variable-free or *ground* expressions:
  - Each ground term denotes an individual in the domain:
    - Constant c denotes the individual $I(c)$ in $\mathcal{I}$.
    - $f(t_1, \ldots, t_n)$ denotes the individual $I(f)(t'_1, \ldots, t'_n)$ in $\mathcal{I}$, where t'_i is the individual denoted by t_i (i.e. $t'_i = I(t_1)$).
  - Each ground atomic formula is either *true* or *false* in an interpretation.
    - Atom $p(t_1, \ldots, t_n)$ is *true* in $\mathcal{I}$ if $\langle t'_1, \ldots, t'_n \rangle \in I(p)$ where $t'_i = I(t_1)$; otherwise *false*.

Semantics (continued)

- We first give a semantics for variable-free or *ground* expressions:
  - Each ground term denotes an individual in the domain:
    - Constant c denotes the individual $I(c)$ in $\mathcal{I}$.
    - $f(t_1, \ldots, t_n)$ denotes the individual $I(f)(t'_1, \ldots, t'_n)$ in $\mathcal{I}$, where t'_i is the individual denoted by t_i (i.e. $t'_i = I(t_1)$).
  - Each ground atomic formula is either *true* or *false* in an interpretation.
    - Atom $p(t_1, \ldots, t_n)$ is *true* in $\mathcal{I}$ if $\langle t'_1, \ldots, t'_n \rangle \in I(p)$ where $t'_i = I(t_1)$; otherwise *false*.
- *Truth* in interpretation $\mathcal{I}$ is defined by:
  - $P \wedge Q$ is true iff P is true and Q is true.
  - $Q \Leftarrow P$ is true iff P is false or Q is true.

☞ At this point every variable-free formula is true or false in an interpretation.

Semantics: Variables

A *variable assignment* ν is used to define the semantics of formulas with variables.

- As with FOL, a variable assignment is a function from the set of variables into the domain.

Semantics: Variables

A *variable assignment* ν is used to define the semantics of formulas with variables.

- As with FOL, a variable assignment is a function from the set of variables into the domain.
- A clause C with variables is false in interpretation $\mathcal{I}$ just if there is a variable assignment ν under which the clause is false.

Semantics: Variables

A *variable assignment* ν is used to define the semantics of formulas with variables.

- As with FOL, a variable assignment is a function from the set of variables into the domain.
- A clause C with variables is false in interpretation $\mathcal{I}$ just if there is a variable assignment ν under which the clause is false.
  - Recall: Variables are local to a clause.
  - Recall: Variables in a clause are regarded as universally quantified.
- A clause C with variables is true in $\mathcal{I}$ just if it isn't false.
  - I.e. C is true for every variable assignment.

Semantics: Entailment

Finally:

- A set of clauses C is *true in an interpretation* $\mathcal{I}$ iff every element of C is true in $\mathcal{I}$.
  - $\mathcal{I}$ is a *model* of C.

Semantics: Entailment

Finally:

- A set of clauses C is *true in an interpretation* $\mathcal{I}$ iff every element of C is true in $\mathcal{I}$.
  - $\mathcal{I}$ is a *model* of C.
- If S is a set of clauses and g is an *atom* or *conjunction of atoms*, then g is *logically entailed* by S, written $S \models g$, iff g is true in every model of S.
  - I.e. every model of S is a model of g.
  - So the same definition as in FOL, but in a restricted language.
  - ☞ Note the restricted form of $\models$.
- The relation $\models$ says nothing about computation, proof, derivation, etc.
  - ☞ $\models$ just says what is true, given that other things are true.

User's View of Semantics

Recall that the idea behind our use of logic is that we have a particular domain in mind to represent, the *intended interpretation*.

- We choose denotations for our symbols with respect to this domain and write, as clauses, what is true in that world.
  - I.e. we *axiomatise* our domain.
- When the system gives us a logical consequence of our axioms we can interpret this answer with respect to our intended interpretation.
- Again, this is no different than in FOL, except that we have a limited language.

Semantics and Logical Consequence

- The computer does not have access to the intended interpretation, but only to the axiomatisation.
- Given an appropriate *inference procedure*, the computer will be able to tell whether some statement is a logical consequence of the axioms.
  - If it is a logical consequence, then it is true in the intended interpretation (assuming the axioms are correct).

Queries and Answers

- As with FOL, we build a formal description of the world in order to ask questions about it.
  - Want to ask about information *implicit* in the knowledge base.
  - If we were just interested in *retrieval* of explicit information (as in a database) we wouldn't need a formal model.
- A *query* defines the syntax by which we ask whether something is a logical consequence of the knowledge base.
- Queries can be represented syntactically as

  $?body.$

Queries and Answers

- A query is a question to which we want the answer:
  - *yes* if the query is a consequence of the knowledge base and
  - *no* if the query is not a consequence of the knowledge base.
- *No* doesn't mean that the query is false in the intended interpretation.
- Rather *no* means that we don't know whether it is true in the intended interpretation.

Queries and Answers

- One way of treating queries, is that for

  ?*body*.

  it is as if we added a clause

  answer $\Leftarrow$ *body*.

  to the knowledge base (for new atom *answer*)
- We then try to show that *answer* is a logical consequence of the KB.
- If we can show that *answer* is a logical consequence, then so is *body*.
- This scheme provides a uniformity wrt query answering; as well it allows us to express *answers* via an *answer predicate* (later).

Variables

- Recall: When a clause contains variables, that clause is true in an interpretation only if it is true for every possible value of the variables.
- So if X appears in clause C then

  C is true in an interpretation

  means that

  C is true no matter what individual is denoted by X.
- For example, for

$$gf(X, Y) \Leftarrow father(X, Z) \wedge parent(Z, Y).$$

  to be true, it must be true no matter what individuals are denoted by X, Y and Z.

Variables

One potentially confusing point is the following:

> *Variables that appear only in the body of a clause can be considered to be universally quantified at the level of the clause, and existentially quantified in the body.*

For example, if we use explicit quantifiers $\forall X$ and $\exists X$, then we have that

$$\forall X \forall Y \forall Z (gf(X, Y) \Leftarrow father(X, Z) \wedge parent(Z, Y))$$

means the same thing as

$$\forall X \forall Y (gf(X, Y) \Leftarrow \exists Z (father(X, Z) \wedge parent(Z, Y)))$$

Variables and Queries

Variables in queries are handled by our previous translation.

- Example: $?gf(X, ian)$ can be translated to:

  $$answer \Leftarrow gf(X, ian)$$

  Or, using the second reading from the previous slide:

  $$answer \Leftarrow \exists X\, gf(X, ian)$$

- I.e *answer* is true if there is some X who is the grandfather of *ian*.

Variables and Queries

- Typically we want to know not just *whether* there is a grandfather of Ian, but *who* the grandfather of Ian is.
- For this we translate the query $?gf(X, ian)$ to the *answer clause*

$$answer(X) \Leftarrow gf(X, ian)$$

- In general, if the query is B with free variables $X_1, \ldots, X_n$, then the answer clause is

$$answer(X_1, \ldots, X_n) \Leftarrow B$$

- The aim now is to determine which *instance* of $answer(X_1, \ldots, X_n)$ is a consequence of the KB.

Inference

- So far we have specified what we would like an answer to be, but not how it can be computed.
  - I.e. we have just considered conditions under which a clause is true in an interpretation.
- Now we want to explore means by which logical consequences of a set of clauses can be computed solely on the basis of their form, and without considering interpretations.
  - I.e. we want to determine an *inference procedure* or *proof procedure* for our clause language.
- For a proof procedure, we write

  $S \vdash g$

  to mean g can be derived from S.

Proof Procedures

- A proof procedure can be judged by whether it computes what it is meant to compute.
- As before:
  - A proof procedure is *sound* with respect to a semantics if everything derivable is justified by the semantics.
    That is
    If $S \vdash g$ *then* $S \models g$.
  - A proof procedure is *complete* with respect to a semantics if there is a proof for every logical consequence of the clauses.
    That is
    If $S \models g$ *then* $S \vdash g$.

A Bottom-up Proof Procedure

- Idea: Starting from the initial facts and rules in the KB, derive further facts.
  - ☞ Also called *forward chaining*.
- The procedure is based on a *rule of derivation*, a generalised rule of "modus ponens":

  If $h \Leftarrow b_1 \wedge \cdots \wedge b_m$ *is a clause, and each* b_i *has been derived, then* h *can be derived.*

- As a base case, we have that every fact is (trivially) derived.
- We consider the variable-free case first.

A Bottom-up Proof Procedure

Procedure:

$C := \{\}$;

repeat

> *choose* $r \in S$ *such that*
> r *is* '$h \Leftarrow b_1 \wedge \cdots \wedge b_m$'
> $b_i \in C$ *for all* i, *and*
> $h \notin C$;
>
> $C := C \cup \{h\}$

until no more choices

We write $S \vdash g$ if $g \in C$ at the end of the procedure.

Example

Example

$$a \Leftarrow b \wedge c$$
$$b \Leftarrow d \wedge e$$
$$c \Leftarrow e$$
$$d$$
$$e$$
$$f \Leftarrow a \wedge g$$

Obtain:

Example

Example

$$a \Leftarrow b \land c$$
$$b \Leftarrow d \land e$$
$$c \Leftarrow e$$
$$d$$
$$e$$
$$f \Leftarrow a \land g$$

Obtain: $\{d, e,$

Example

Example

$a \Leftarrow b \wedge c$
$b \Leftarrow d \wedge e$
$c \Leftarrow e$
d
e
$f \Leftarrow a \wedge g$

Obtain: $\{d, e, c,$

Example

Example

$$a \Leftarrow b \wedge c$$
$$b \Leftarrow d \wedge e$$
$$c \Leftarrow e$$
$$d$$
$$e$$
$$f \Leftarrow a \wedge g$$

Obtain: $\{d, e, c, b,$

Example

Example

$$a \Leftarrow b \wedge c$$
$$b \Leftarrow d \wedge e$$
$$c \Leftarrow e$$
$$d$$
$$e$$
$$f \Leftarrow a \wedge g$$

Obtain: $\{d, e, c, b, a\}$.

Properties of the Procedure:

1. Soundness: Every atom in C is a logical consequence of S.

Properties of the Procedure:

1. Soundness: Every atom in C is a logical consequence of S.
2. Completeness: If $S \models g$ then $S \vdash g$.
   - ☞ This just applies to atoms (and not clauses in general).

Properties of the Procedure:

1. Soundness: Every atom in C is a logical consequence of S.
2. Completeness: If $S \models g$ then $S \vdash g$.
   ☞ This just applies to atoms (and not clauses in general).
3. Complexity: The algorithm halts and the number of iterations is bounded by the number of clauses in S.
   - The algorithm is linear in the size of the KB (provided we index the clauses so that the inside loop can be carried out in constant time).

Properties of the Procedure:

1. Soundness: Every atom in C is a logical consequence of S.
2. Completeness: If $S \models g$ then $S \vdash g$.
   ☞ This just applies to atoms (and not clauses in general).
3. Complexity: The algorithm halts and the number of iterations is bounded by the number of clauses in S.
   - The algorithm is linear in the size of the KB (provided we index the clauses so that the inside loop can be carried out in constant time).
4. Fixed Point: The final C is called a *fixed point*.
   - Let $\mathcal{I}$ be the interpretation in which every atom in the fixed point is *true* and every atom not in the fixed point is *false*. Then: $\mathcal{I}$ is a model of S.

Properties of the Procedure:

1. Soundness: Every atom in C is a logical consequence of S.
2. Completeness: If $S \models g$ then $S \vdash g$.
   ☞ This just applies to atoms (and not clauses in general).
3. Complexity: The algorithm halts and the number of iterations is bounded by the number of clauses in S.
   - The algorithm is linear in the size of the KB (provided we index the clauses so that the inside loop can be carried out in constant time).
4. Fixed Point: The final C is called a *fixed point*.
   - Let $\mathcal{I}$ be the interpretation in which every atom in the fixed point is *true* and every atom not in the fixed point is *false*. Then: $\mathcal{I}$ is a model of S.

Exercise: Prove the above items.

A Top-down Proof Procedure

An alternative proof method is to search backwards from the query to determine whether it is a logical consequence of S.

☞ Also called *backward-chaining* inference

A Top-down Proof Procedure

An alternative proof method is to search backwards from the query to determine whether it is a logical consequence of S.

☞ Also called *backward-chaining* inference

- We define *definite clause resolution* for the ground case, then consider the general case with variables.

A Top-down Proof Procedure

An alternative proof method is to search backwards from the query to determine whether it is a logical consequence of S.

☞ Also called *backward-chaining* inference

- We define *definite clause resolution* for the ground case, then consider the general case with variables.
- An *answer clause* is of the form

$$answer \Leftarrow a_1 \wedge \cdots \wedge a_m$$

A Top-down Proof Procedure

An alternative proof method is to search backwards from the query to determine whether it is a logical consequence of S.

☞ Also called *backward-chaining* inference

- We define *definite clause resolution* for the ground case, then consider the general case with variables.
- An *answer clause* is of the form

$$answer \Leftarrow a_1 \wedge \cdots \wedge a_m$$

- A *resolution* of the above clause with the clause

$$a_1 \Leftarrow b_1 \wedge \cdots \wedge b_n \qquad \text{is the answer clause}$$

$$answer \Leftarrow b_1 \wedge \cdots \wedge b_n \wedge a_2 \wedge \cdots \wedge a_m$$

A Top-down Proof Procedure

An alternative proof method is to search backwards from the query to determine whether it is a logical consequence of S.

☞ Also called *backward-chaining* inference

- We define *definite clause resolution* for the ground case, then consider the general case with variables.
- An *answer clause* is of the form

$$answer \Leftarrow a_1 \wedge \cdots \wedge a_m$$

- A *resolution* of the above clause with the clause

$$a_1 \Leftarrow b_1 \wedge \cdots \wedge b_n \qquad \text{is the answer clause}$$

$$answer \Leftarrow b_1 \wedge \cdots \wedge b_n \wedge a_2 \wedge \cdots \wedge a_m$$

- An *answer* is an answer clause with no body

A Top-down Proof Procedure

- A *derivation* of a query $?q_1 \wedge \cdots \wedge q_k$ from rules S is a sequence of answer clauses $\gamma_0, \ldots, \gamma_p$ such that
  1. γ_0 is the answer clause:

     $$answer \Leftarrow q_1 \wedge \cdots \wedge q_k,$$

  2. γ_i is obtained by resolving γ_{i-1} with a clause in S, and
  3. γ_p is an answer.

A Top-down Proof Procedure

- A *derivation* of a query $?q_1 \wedge \cdots \wedge q_k$ from rules S is a sequence of answer clauses $\gamma_0, \ldots, \gamma_p$ such that
  1. γ_0 is the answer clause:
  $$answer \Leftarrow q_1 \wedge \cdots \wedge q_k,$$
  2. γ_i is obtained by resolving γ_{i-1} with a clause in S, and
  3. γ_p is an answer.
- This is just proposition resolution under a (slightly) different guise and in a simpler language.
- Note that it implements a *set of support* strategy.

A Top-down Interpreter:

solve($q_1 \wedge \cdots \wedge q_k$):

ac $:= \{answer \Leftarrow q_1 \wedge \cdots \wedge q_k\}$

repeat

  choose C from S

  $ac := resolve(ac, C)$

until ac is an answer

- Note that in this case, the nondeterministic "choose" relies on guessing the "right" clause for resolution.
- The differing types of nondeterminism (as in the bottom-up and top-down procedures) have been called *select* vs. *choose* nondeterminism.

Aside: Select and Choose Nondeterminism

Select nondeterminism:

- For *select* nondeterminism, if the language is finite and there are no variables, then it doesn't matter what nondeterministic choice you make.
- E.g. for the bottom-up procedure, eventually every derivable atom will be derived.
- For variables you have to be more careful.

Choose nondeterminism:

- For *choose* nondeterminism, one has to make the "right" nondeterministic choice.
- Just because one choice doesn't lead to an answer doesn't mean other choices will be futile.
- So here we also have a search problem.

Example:

Example

$a \Leftarrow b \wedge c$
$b \Leftarrow d \wedge e$
$c \Leftarrow e$
d
e
$f \Leftarrow a \wedge g$
$?a$

One sequence of assignments to *answer* is:
$answer \Leftarrow a$

Example:

Example

$a \Leftarrow b \wedge c$
$b \Leftarrow d \wedge e$
$c \Leftarrow e$
d
e
$f \Leftarrow a \wedge g$
$?a$

One sequence of assignments to *answer* is:

$answer \Leftarrow a$
$answer \Leftarrow b \wedge c$

Example:

Example

$a \Leftarrow b \wedge c$

$b \Leftarrow d \wedge e$

$c \Leftarrow e$

d

e

$f \Leftarrow a \wedge g$

$?a$

One sequence of assignments to *answer* is:

$answer \Leftarrow a$

$answer \Leftarrow b \wedge c$

$answer \Leftarrow d \wedge e \wedge c$

Example:

Example

$a \Leftarrow b \wedge c$
$b \Leftarrow d \wedge e$
$c \Leftarrow e$
d
e
$f \Leftarrow a \wedge g$
$?a$

One sequence of assignments to *answer* is:

$answer \Leftarrow a$
$answer \Leftarrow b \wedge c$
$answer \Leftarrow d \wedge e \wedge c$
$answer \Leftarrow e \wedge c$

Example:

Example

$a \Leftarrow b \wedge c$
$b \Leftarrow d \wedge e$
$c \Leftarrow e$
d
e
$f \Leftarrow a \wedge g$
$?a$

One sequence of assignments to *answer* is:

$answer \Leftarrow a$
$answer \Leftarrow b \wedge c$
$answer \Leftarrow d \wedge e \wedge c$
$answer \Leftarrow e \wedge c$
$answer \Leftarrow c$

Example:

Example

$a \Leftarrow b \wedge c$

$b \Leftarrow d \wedge e$

$c \Leftarrow e$

d

e

$f \Leftarrow a \wedge g$

$?a$

One sequence of assignments to *answer* is:

$answer \Leftarrow a$

$answer \Leftarrow b \wedge c$

$answer \Leftarrow d \wedge e \wedge c$

$answer \Leftarrow e \wedge c$

$answer \Leftarrow c$

$answer \Leftarrow e$

Example:

Example

$$a \Leftarrow b \wedge c$$
$$b \Leftarrow d \wedge e$$
$$c \Leftarrow e$$
$$d$$
$$e$$
$$f \Leftarrow a \wedge g$$
$$?a$$

One sequence of assignments to *answer* is:

$$answer \Leftarrow a$$
$$answer \Leftarrow b \wedge c$$
$$answer \Leftarrow d \wedge e \wedge c$$
$$answer \Leftarrow e \wedge c$$
$$answer \Leftarrow c$$
$$answer \Leftarrow e$$
$$answer \Leftarrow$$

Notes

1. When we have derived the answer, we can read a bottom-up "proof" in the opposite direction.
   - Also every top-down derivation corresponds to a bottom-up proof and every bottom-up proof has a corresponding top-down derivation.
2. The preceding equivalence can be used to show the soundness and completeness of the derivation procedure.

Variables and Substitutions

Variables and substitutions are handled exactly as in FOL:

- An *instance* of a clause is obtained by uniformly substituting terms for variables in the clause.
- If a clause is true in an interpretation then any instance will also be true in that interpretation.
- A *substitution* is a set of statements of the form v/t, where v is a variable and t is a term.

Problem: There may be infinitely many instances of a clause if we have function symbols.

- E.g.: $num(0)$, $num(s(0))$, $num(s(s(0)))$, ...

Variables and Substitutions

- A substitution is in *normal form* if each variable on the left-hand side appears nowhere else in the substitution.
  - Assume all substitutions are in normal form.
- A substitution θ *applied* to an expression e is an expression $e\theta$ which is like e, but with all instances of variables on the lhs of a "/" replaced by the term on the rhs.
- E.g., applying

  $$\theta = \{X/Y, Z/f(U)\}$$

  to

  $$p(X, Y) \Leftarrow q(a, Z).$$

  is the instance

  $$p(Y, Y) \Leftarrow q(a, f(U)).$$

Variables and Substitutions

Recall:

- Substitution θ is a *unifier* of atoms e_1 and e_2 if $e_1\theta = e_2\theta$.
  - E.g. $\{X/a, Y/b\}$ is a unifier of $t(a, Y, c)$ and $t(X, b, c)$.

Variables and Substitutions

Recall:

- Substitution θ is a *unifier* of atoms e_1 and e_2 if $e_1\theta = e_2\theta$.
  - E.g. $\{X/a, Y/b\}$ is a unifier of $t(a, Y, c)$ and $t(X, b, c)$.
- There may be many unifiers for terms and clauses.
  - E.g, $p(X, Y)$ and $p(Z, Z)$ have unifiers
    $\{X/b,\ Y/b,\ Z/b\}$
    $\{X/f(a),\ Y/f(a),\ Z/f(a)\}$
    $\{X/Z,\ Y/Z\}$.

Variables and Substitutions

Recall:

- Substitution θ is a *unifier* of atoms e_1 and e_2 if $e_1\theta = e_2\theta$.
  - E.g. $\{X/a, Y/b\}$ is a unifier of $t(a, Y, c)$ and $t(X, b, c)$.
- There may be many unifiers for terms and clauses.
  - E.g, $p(X, Y)$ and $p(Z, Z)$ have unifiers
    $\{X/b,\ Y/b,\ Z/b\}$
    $\{X/f(a),\ Y/f(a),\ Z/f(a)\}$
    $\{X/Z,\ Y/Z\}$.
- The third unifier is preferred because it implies the first two.
  - This is called the *most general unifier*, or MGU.
  - So the MGU is a unifier of two terms that is implied by all other unifiers.
- MGU's exist and are unique, up to the renaming of variables.

Bottom-up Procedure with Variables

- We can do the bottom-up procedure for clauses with variables, if we carry out the bottom-up procedure for all ground instances of the variables in the axioms.
- We must make certain that our procedure is *fair*, in that every usable rule is chosen eventually.
- E.g., consider:
  $num(s(N)) \Leftarrow num(N)$
  $num(0)$
  $mother(sue, mary).$

  An unfair strategy could always choose the first rule, and so never derive that $mother(sue, mary)$.
- Our previous procedure, extended to allow variables, is sound and complete (so long as it is fair).

Bottom-up Procedure with Variables

If the domain is known to be finite, then one can handle variables by:

1. Substitute all possible instances of terms for the variables in the KB.
   - This is known as *grounding* the KB.
2. Then work with the grounded KB, using the procedure for propositional KBs.

☞ Thus the first-order set of rules is effectively translated into a KB in propositional logic.

Top-down Procedure with Variables

Or: *Definite clause resolution with variables*.

- Suppose we have the answer clause

  $$answer(t_1, \ldots, t_k) \Leftarrow a_1 \wedge \cdots \wedge a_m$$

- The *resolution* of the above clause with the clause

  $$a \Leftarrow b_1 \wedge \cdots \wedge b_n$$

  where a and a_1 have most general unifier θ is the answer clause:

  $$[answer(t_1, \ldots, t_k) \Leftarrow b_1 \wedge \cdots \wedge b_n \wedge a_2 \wedge \cdots \wedge a_m]\theta$$

- This is known as *SLD resolution*
- SLD resolution is the principal control strategy that underlies PROLOG.

Definite clause resolution with variables

- A *derivation* from rules S is a sequence of answer clauses $\gamma_0, \ldots, \gamma_n$ such that
  1. γ_0 is the original answer clause.
     If the query is B with free variables $V_1, \ldots, V_k$, then γ_0 is
     $$answer(V_1, \ldots, V_k) \Leftarrow B.$$
  2. γ_i is obtained by resolving γ_{i-1} with a clause in S.
  3. γ_n is an answer.
     - That is, γ_n is of the form

       $$answer(t_1, \ldots, t_k) \Leftarrow .$$

     When this occurs we have an answer, $(V_1 = t_1, \ldots, V_k = t_k)$.

Example

(from before):

$$gf(X,Y) \Leftarrow father(X,Z) \wedge parent(Z,Y)$$
$$parent(X,Y) \Leftarrow mother(X,Y)$$
$$parent(X,Y) \Leftarrow father(X,Y)$$
$$mother(michelle, sue)$$
$$father(ian, sue)$$
$$mother(sue, chris)$$
$$father(george, ian)$$

Example

For query $?gf(G, sue)$, we have the derivation:

1. $answer(G) \Leftarrow gf(G, sue)$

Example

For query $?gf(G, sue)$, we have the derivation:

1. $answer(G) \Leftarrow gf(G, sue)$
   This is resolved with the first clause in the KB with substitution $\{X_1/G, Y_1/sue\}$ to obtain
2. $answer(G) \Leftarrow father(G, Z_1) \wedge parent(Z_1, sue)$

Example

For query $?gf(G, sue)$, we have the derivation:

1. $answer(G) \Leftarrow gf(G, sue)$
   This is resolved with the first clause in the KB with substitution $\{X_1/G, Y_1/sue\}$ to obtain
2. $answer(G) \Leftarrow father(G, Z_1) \wedge parent(Z_1, sue)$
   This is resolved with $father(george, ian)$ with substitution $\{G/george, Z_1/ian\}$ to obtain
3. $answer(george) \Leftarrow parent(ian, sue)$

Example

For query $?gf(G, sue)$, we have the derivation:

1. $answer(G) \Leftarrow gf(G, sue)$
   This is resolved with the first clause in the KB with substitution $\{X_1/G, Y_1/sue\}$ to obtain
2. $answer(G) \Leftarrow father(G, Z_1) \wedge parent(Z_1, sue)$
   This is resolved with $father(george, ian)$ with substitution $\{G/george, Z_1/ian\}$ to obtain
3. $answer(george) \Leftarrow parent(ian, sue)$
   This is resolved with $parent(X_2, Y_2) \Leftarrow father(X_2, Y_2)$ with substitution $\{X_2/ian, Y_2/sue\}$ to obtain
4. $answer(george) \Leftarrow father(ian, sue)$

Example

For query $?gf(G, sue)$, we have the derivation:

1. $answer(G) \Leftarrow gf(G, sue)$
   This is resolved with the first clause in the KB with substitution $\{X_1/G, Y_1/sue\}$ to obtain
2. $answer(G) \Leftarrow father(G, Z_1) \wedge parent(Z_1, sue)$
   This is resolved with $father(george, ian)$ with substitution $\{G/george, Z_1/ian\}$ to obtain
3. $answer(george) \Leftarrow parent(ian, sue)$
   This is resolved with $parent(X_2, Y_2) \Leftarrow father(X_2, Y_2)$ with substitution $\{X_2/ian, Y_2/sue\}$ to obtain
4. $answer(george) \Leftarrow father(ian, sue)$
   This is resolved with $father(ian, sue)$ to obtain
5. $answer(george) \Leftarrow$

An answer thus is $G = george$.

Example

Notes:

- Another answer could have been chosen by choosing different clauses for resolution.
- Some choice of clauses for resolution will lead to a dead end.
- There is an (implicit) renaming of variables for each instance/use of a clause.
- A full implementation will need to save state information in order to determine another answer.

Example: Simulating Systems

Example

Consider the domain of circuits.

- We have objects consisting of *gates* of various types, *signal values* (i.e. *on* and *off*), etc.
- We use the following predicates and functions:
  1. $gate(G, T)$ means that gate G is of type T.
     E.g.: $gate(x_1, xor)$, $gate(x_2, xor)$, $gate(a_1, and)$, $gate(a_2, and)$, $gate(o_1, or)$.
  2. $Connected(P_1, P_2)$ means that port P_1 is connected to port P_2.
  3. $in(N, G)$ denotes input port N of gate G.
  4. $out(G)$ denotes the output port of gate G.
  5. $out(N, G)$ denotes output port N of circuit G.

Example: Simulating Systems

- For connectivity we can assert something like:

$$value(X, V) \Leftarrow connected(Y, X) \wedge value(Y, V).$$

- To say that an *and* gate has output corresponding to the conjunction of its inputs we could have:

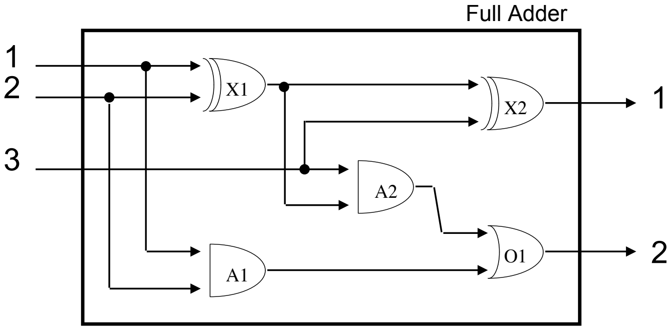
$$\begin{aligned} value(out(D), on) \quad \Leftarrow \quad & gate(D, and) \\ & \wedge\, value(in(1, D), on) \\ & \wedge\, value(in(2, D), on). \end{aligned}$$

$$value(out(D), off) \Leftarrow gate(D, and) \wedge value(in(1, D), off).$$

$$value(out(D), off) \Leftarrow gate(D, and) \wedge value(in(2, D), off).$$

Example: Simulating Systems

Consider a full adder:



Example: Simulating Systems

- We can add assertions about the values of the inputs to the circuits such as

$$value(in(1, adder), on),$$
$$value(in(2, adder), off),$$
$$value(in(3, adder), on)$$

- We can determine the values of the output ports with the query

$$?value(out(1, adder), Out1) \wedge value(out(2, adder), Out2)$$

- This returns $Out1 = off$ and $Out2 = on$.

Bottom-Up vs. Top-Down Derivations

Ask: why select top-down procedure over bottom-up, or vice versa?

- Top-down/Backward Chaining:
  - Query-answering
  - Directed reasoning
  - Good for user acceptability and diagnosis of KB bugs.
  - Worst-case exponential complexity
  - Harder to implement
- Bottom-up/Forward Chaining:
  - Gives all solutions
  - More responsive to changes in domain facts
    - E.g. Rules of form: Action $\Leftarrow$ Condition
  - Linear procedure
  - More suitable for finite domains.
  - With variables, typically need to *ground* the knowledge base first