

For Wednesday

- No reading
- Chapter 9, exercise 9
 - Must be proper Horn clauses

Same Variable

- Exact variable names used in sentences in the KB **should** not matter.
- But if **Likes(x,FOPC)** is a formula in the KB, it does not unify with **Likes(John,x)** but does unify with **Likes(John,y)**
- We can standardize one of the arguments to UNIFY to make its variables unique by renaming them.

Likes(x,FOPC) \rightarrow Likes(x_1 , FOPC)

UNIFY(Likes(John,x),Likes(x_1 ,FOPC)) = { x_1 /John, x/FOPC }

Which Unifier?

- There are many possible unifiers for some atomic sentences.
 - $\text{UNIFY}(\text{Likes}(x,y),\text{Likes}(z,\text{FOPC})) =$
 - $\{x/z, y/\text{FOPC}\}$
 - $\{x/\text{John}, z/\text{John}, y/\text{FOPC}\}$
 - $\{x/\text{Fred}, z/\text{Fred}, y/\text{FOPC}\}$
 -
- UNIFY should return the most general unifier which makes the least commitment to variable values.

How Do We Use It?

- We have two primary methods for using Generalized Modus Ponens
- We can start with the knowledge base and try to generate new sentences
 - Forward Chaining
- We can start with a sentence we want to prove and try to work backward until we can establish the facts from the knowledge base
 - Backward Chaining

Forward Chaining

- Use modus ponens to derive **all** consequences from new information.
- Inferences cascade to draw deeper and deeper conclusions
- To avoid looping and duplicated effort, must prevent addition of a sentence to the KB which is the same as one already present.
- Must determine all ways in which a rule (Horn clause) can match existing facts to draw new conclusions.

Assumptions

- A sentence is a renaming of another if it is the same except for a renaming of the variables.
- The composition of two substitutions combines the variable bindings of both such that:

$$\text{SUBST}(\text{COMPOSE}(\theta_1, \theta_2), p) = \text{SUBST}(\theta_2, \text{SUBST}(\theta_1, p))$$

Forward Chaining Algorithm

procedure FORWARD-CHAIN(KB, p)

if there is a sentence in KB that is a renaming of p then return

Add p to KB

for each $(p_1 \wedge \dots \wedge p_n \Rightarrow q)$ in KB such that for some i ,

UNIFY(p_i, p) = θ succeeds do

FIND-AND-INFER($KB, [p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n], q, \theta$)

end

procedure FIND-AND-INFER($KB, premises, conclusion, \theta$)

if $premises = []$ then

FORWARD-CHAIN($KB, SUBST(\theta, conclusion)$)

else for each p' in KB such that

UNIFY($p', SUBST(\theta, FIRST(premises))$) = θ_2 do

FIND-AND-INFER($KB, REST(premises), conclusion, COMPOSE(\theta, \theta_2)$)

end

Forward Chaining Example

Assume in KB

1) $\text{Parent}(x,y) \wedge \text{Male}(x) \Rightarrow \text{Father}(x,y)$

2) $\text{Father}(x,y) \wedge \text{Father}(x,z) \Rightarrow \text{Sibling}(y,z)$

Add to KB

3) $\text{Parent}(\text{Tom}, \text{John})$

Rule 1) tried but can't "fire"

Add to KB

4) $\text{Male}(\text{Tom})$

Rule 1) now satisfied and triggered and adds:

5) $\text{Father}(\text{Tom}, \text{John})$

Rule 2) now triggered and adds:

6) $\text{Sibling}(\text{John}, \text{John}) \{ x/\text{Tom}, y/\text{John}, z/\text{John} \}$

Example cont.

Add to KB

7) Parent(Tom,Fred)

Rule 1) triggered again and adds:

8) Father(Tom,Fred)

Rule 2) triggered again and adds:

9) Sibling(Fred,Fred) { x/Tom, y/Fred, z/Fred }

Rule 2) triggered again and adds:

10) Sibling(John, Fred) { x/Tom, y/John, z/Fred }

Rule 2) triggered again and adds:

11) Sibling(Fred, John) { x/Tom, y/Fred, z/John }

Problems with Forward Chaining

- Inference can explode forward and may never terminate.
- Consider the following:
 - $\text{Even}(x) \Rightarrow \text{Even}(\text{plus}(x,2))$
 - $\text{Integer}(x) \Rightarrow \text{Even}(\text{times}(2,x))$
 - $\text{Even}(x) \Rightarrow \text{Integer}(x)$
 - $\text{Even}(2)$
- Inference is not directed towards any particular conclusion or goal. May draw lots of irrelevant conclusions

Backward Chaining

- Start from query or atomic sentence to be proven and look for ways to prove it.
- Query can contain variables which are assumed to be existentially quantified.

Sibling(x,John) ?

Father(x,y) ?

- Inference process should return all sets of variable bindings that satisfy the query.

Method

- First try to answer query by unifying it to all possible facts in the KB.
- Next try to prove it using a rule whose consequent unifies with the query and then try to recursively prove all of its antecedents.
- Given a conjunction of queries, first get all possible answers to the first conjunct and then for each resulting substitution try to prove all of the remaining conjuncts.
- Assume variables in rules are renamed (standardized apart) before each use of a rule.

Backchaining Examples

KB:

- 1) $\text{Parent}(x,y) \wedge \text{Male}(x) \Rightarrow \text{Father}(x,y)$
- 2) $\text{Father}(x,y) \wedge \text{Father}(x,z) \Rightarrow \text{Sibling}(y,z)$
- 3) $\text{Parent}(\text{Tom},\text{John})$
- 4) $\text{Male}(\text{Tom})$
- 7) $\text{Parent}(\text{Tom},\text{Fred})$

Query: $\text{Parent}(\text{Tom},x)$

Answers: ($\{x/\text{John}\}, \{x/\text{Fred}\}$)

Query: $\text{Father}(\text{Tom}, s)$

Subgoal: $\text{Parent}(\text{Tom}, s) \wedge \text{Male}(\text{Tom})$

$\{s/\text{John}\}$

Subgoal: $\text{Male}(\text{Tom})$

Answer: $\{s/\text{John}\}$

$\{s/\text{Fred}\}$

Subgoal: $\text{Male}(\text{Tom})$

Answer: $\{s/\text{Fred}\}$

Answers: $(\{s/\text{John}\}, \{s/\text{Fred}\})$

Query: $\text{Father}(f,s)$

Subgoal: $\text{Parent}(f,s) \wedge \text{Male}(f)$

$\{f/\text{Tom}, s/\text{John}\}$

Subgoal: $\text{Male}(\text{Tom})$

Answer: $\{f/\text{Tom}, s/\text{John}\}$

$\{f/\text{Tom}, s/\text{Fred}\}$

Subgoal: $\text{Male}(\text{Tom})$

Answer: $\{f/\text{Tom}, s/\text{Fred}\}$

Answers: $(\{f/\text{Tom}, s/\text{John}\}, \{f/\text{Tom}, s/\text{Fred}\})$

Query: Sibling(a,b)

Subgoal: Father(f,a) \wedge Father(f,b)

{f/Tom, a/John}

Subgoal: Father(Tom,b)

{b/John}

Answer: {f/Tom, a/John, b/John}

{b/Fred}

Answer: {f/Tom, a/John, b/Fred}

{f/Tom, a/Fred}

Subgoal: Father(Tom,b)

{b/John}

Answer: {f/Tom, a/Fred, b/John}

{b/Fred}

Answer: {f/Tom, a/Fred, b/Fred}

Answers: ({f/Tom, a/John, b/John}, {f/Tom, a/John, b/Fred})

{f/Tom, a/Fred, b/John}, {f/Tom, a/Fred, b/Fred})

Incompleteness

- Rule-based inference is not complete, but is reasonably efficient and useful in many circumstances.
- Still can be exponential or not terminate in worst case.

- Incompleteness example:

$$P(x) \Rightarrow Q(x)$$

$$\neg P(x) \Rightarrow R(x) \text{ (not Horn)}$$

$$Q(x) \Rightarrow S(x)$$

$$R(x) \Rightarrow S(x)$$

- Entails $S(A)$ for any constant A but is not inferable from modus ponens

Completeness

- In 1930 Gödel showed that a complete inference procedure for FOPC existed, but did not demonstrate one (non-constructive proof).
- In 1965, Robinson showed a resolution inference procedure that was sound and complete for FOPC.
- However, the procedure may not halt if asked to prove a theorem that is not true, it is said to be semidecidable (a type of undecidability).
- If a conclusion C is entailed by the KB then the procedure will eventually terminate with a proof. However if it is not entailed, it may never halt.
- It does not follow that either C or $\neg C$ is entailed by a KB (may be independent). Therefore trying to prove both a conjecture and its negation does not help.
- Inconsistency of a KB is also semidecidable.

Logic Programming

- Also called **declarative programming**
- We write programs that say what is to be the result
- We don't specify how to get the result
- Based on logic, specifically first order predicate calculus

Prolog

- **Programming in Logic**
- Developed in 1970's
- ISO standard published in 1996
- Used for:
 - Artificial Intelligence: expert systems, natural language processing, machine learning, constraint satisfaction, anything with rules
 - Logic databases
 - Prototyping

Bibliography

- Clocksin and Mellish, Programming in Prolog
- Bratko, Prolog Programming for Artificial Intelligence
- Sterling and Shapiro, The Art of Prolog
- O'Keefe, The Craft of Prolog

Working with Prolog

- You interact with the Prolog **listener**.
- Normally, you operate in a querying mode which produces **backward chaining**.
- New facts or rules can be entered into the Prolog database either by **consulting** a file or by switching to **consult** mode and typing them into the listener.

Prolog and Logic

- First order logic with different syntax
- Horn clauses
- Does have extensions for math and some efficiency.

The parent Predicate

- Definition of parent/2 (uses facts only)

```
%parent(Parent,Child).
```

```
parent(pam, bob).
```

```
parent(tom, liz).
```

```
parent(bob, ann).
```

```
parent(bob, pat).
```

```
parent(pat, jim).
```

Constants in Prolog

- Two kinds of constants:
 - Numbers (much like numbers in other languages)
 - Atoms
 - Alphanumeric strings which begin with a lowercase letter
 - Strings of special characters (usually used as operators)
 - Strings of characters enclosed in single quotes

Variables in Prolog

- Prolog variables begin with capital letters.
- We make queries by using variables:
 ?- parent(bob,X).
 X = ann
- Prolog variables are **logic** variables, **not** containers to store values in.
- Variables become **bound** to their values.
- The answers from Prolog queries reflect the bindings.

Query Resolution

- When given a query, Prolog tries to find a fact or rule which matches the query, binding variables appropriately.
- It starts with the first fact or rule listed for a given predicate and goes through the list in order.
- If no match is found, Prolog returns no.

Backtracking

- We can get multiple answers to a single Prolog query if multiple items match:
?- parent(X,Y).
- We do this by typing a semi-colon after the answer.
- This causes Prolog to backtrack, unbinding variables and looking for the next match.
- Backtracking also occurs when Prolog attempts to satisfy rules.

Rules in Prolog

- Example Prolog Rule:

```
offspring(Child, Parent) :-  
    parent(Parent, Child).
```

- You can read “:-” as “if”
- Variables with the same name must be bound to the same thing.

Rules in Prolog

- Suppose we have a set of facts for male/1 and female/1 (such as female(ann).).
- We can then define a rule for mother/2 as follows:

```
mother(Mother, Child) :-  
    parent(Mother, Child),  
    female(Mother).
```

- The comma is the Prolog symbol for **and**.
- The semi-colon is the Prolog symbol for **or**.

Recursive Predicates

- Consider the notion of an ancestor.
- We can define a predicate, ancestor/2, using parent/2 if we make ancestor/2 recursive.

Lists in Prolog

- The empty list is represented as [].
- The first item is called the head of the list.
- The rest of the list is called the tail.

List Notation

- We write a list as: $[a, b, c, d]$
- We can indicate the tail of a list using a vertical bar:

$$L = [a, b, c, d],$$

$$L = [\text{Head} \mid \text{Tail}],$$

$$L = [H1, H2 \mid T].$$

$$\text{Head} = a, \text{Tail} = [b, c, d],$$

$$H1 = a, H2 = b, T = [c, d]$$

Some List Predicates

- member/2
- append/3

Try It

- `reverse(List,ReversedList)`
- `evenlength(List)`
- `oddlength(List)`

The Anonymous Variable

- Some variables only appear once in a rule
- Have no relationship with anything else
- Can use `_` for each such variable

Arithmetic in Prolog

- Basic arithmetic operators are provided for by built-in procedures:

$+$, $-$, $*$, $/$, `mod`, `//`

- Note carefully:

`?- X = 1 + 2.`

`X = 1 + 2`

`?- X is 1 + 2.`

`X = 3`

Arithmetic Comparison

- Comparison operators:

>

<

>=

=< (note the order: NOT <=)

== (equal values)

!= (not equal values)

Arithmetic Examples

- Retrieving people born 1950-1960:
?- born(Name, Year),
Year >= 1950,
Year =< 1960.
- Difference between = and ::==
?- 1 + 2 ::= 2 + 1.
yes
?- 1 + 2 = 2 + 1.
no
?- 1 + A = B + 2.
A = 2
B = 1

Length of a List

- Definition of length/2

length([], 0).

length([_ | Tail], N) :-
 length(Tail, N1),
 N is 1 + N1.

- Note: all loops must be implemented via recursion

Counting Loops

- Definition of sum/3

sum(Begin, End, Sum) :-

 sum(Begin, End, Begin, Sum).

sum(X, X, Y, Y).

sum(Begin, End, Sum1, Sum) :-

 Begin < End,

 Next is Begin + 1,

 Sum2 is Sum1 + Next,

 sum(Next, End, Sum2, Sum).

The Cut (!)

- A way to prevent backtracking.
- Used to simplify and to improve efficiency.

Negation

- Can't say something is NOT true
- Use a **closed world assumption**
- Not simply means “I can't prove that it is true”

Dynamic Predicates

- A way to write self-modifying code, in essence.
- Typically just storing data using Prolog's built-in predicate database.
- Dynamic predicates must be declared as such.

Using Dynamic Predicates

- assert and variants
- retract
 - Fails if there is no clause to retract
- retractall
 - Doesn't fail if no clauses