Lecture 14: Forward and Backward Chaining in FOL
Reading: AIAMA 9.3-9.4

Today’s Schedule:
- Review FOL and unification
- Define definite clauses
- Forward Chaining
- Backward Chaining and PROLOG
Recall the Unification Algorithm

function \textsc{Unify}(x, y, \theta) \textbf{returns} a substitution to make \( x \) and \( y \) identical

\textbf{inputs:} \( x \), a variable, constant, list, or compound expression
\( y \), a variable, constant, list, or compound expression
\( \theta \), the substitution built up so far (optional, defaults to empty)

if \( \theta = \text{failure} \) then \textbf{return} failure
else if \( x = y \) then \textbf{return} \( \theta \)
else if \( \text{Variable}?(x) \) then \textbf{return} \textsc{Unify-VAR}(x, y, \theta)
else if \( \text{Variable}?(y) \) then \textbf{return} \textsc{Unify-VAR}(y, x, \theta)
else if \( \text{Compound}?(x) \) and \( \text{Compound}?(y) \) then
  \textbf{return} \textsc{Unify}(x.\text{ARGS}, y.\text{ARGS}, \textsc{Unify}(x.\text{OP}, y.\text{OP}, \theta))
else if \( \text{List}?(x) \) and \( \text{List}?(y) \) then
  \textbf{return} \textsc{Unify}(x.\text{REST}, y.\text{REST}, \textsc{Unify}(x.\text{FIRST}, y.\text{FIRST}, \theta))
else \textbf{return} failure

function \textsc{Unify-Var}(\text{var}, x, \theta) \textbf{returns} a substitution

if \{ \text{var}/val \} \in \theta \text{ then } \textbf{return} \textsc{Unify}(\text{val}, x, \theta)
else if \{ x/val \} \in \theta \text{ then } \textbf{return} \textsc{Unify}(\text{var}, \text{val}, \theta)
else if \( \text{Occur\text{-}Check}?(\text{var}, x) \) then \textbf{return} failure
else \textbf{return} add \{ \text{var}/x \} \text{ to } \theta
Unification Examples

In the following: \( P \) is a predicate, \( f \) and \( g \) are functions, \( A, B \) are constants, \( u, v, x, y, z \) are variables. What is the MGU of sentences \( S_1 \) and \( S_2 \)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>( S_2 )</td>
<td>( \theta )</td>
</tr>
<tr>
<td>( P(x) )</td>
<td>( P(A) )</td>
<td></td>
</tr>
<tr>
<td>( P(f(x), y, g(x)) )</td>
<td>( P(f(x), x, g(x)) )</td>
<td></td>
</tr>
<tr>
<td>( P(f(x), y, g(y)) )</td>
<td>( P(f(x), z, g(x)) )</td>
<td></td>
</tr>
<tr>
<td>( P(x, B, B) )</td>
<td>( P(A, y, z) )</td>
<td></td>
</tr>
<tr>
<td>( P(g(f(v)), g(u)) )</td>
<td>( P(x, x) )</td>
<td></td>
</tr>
<tr>
<td>( P(x, f(x)) )</td>
<td>( P(y, y) )</td>
<td></td>
</tr>
</tbody>
</table>
Forward Chaining in FOL

- FOL definite clauses are disjunctions of literals where exactly one is positive: Situation $\implies$ Result
- FOL forward chaining is iterated application of lifted modus ponens
Examples of Datalog Knowledge Bases

- The classic example is ancestry
- Consider the lifted digital logic
- What can’t be defined this way?
  Example: blocks world
The Forward Chaining Algorithm

function FOL-FC-Ask(KB, α) returns a substitution or false
inputs: KB, the knowledge base, a set of first-order definite clauses
        α, the query, an atomic sentence
local variables: new, the new sentences inferred on each iteration

repeat until new is empty
    new ← { }
    for each rule in KB do
        (p₁ ∧ … ∧ pₙ → q) ← STANDARDIZE-VARIABLES(rule)
        for each θ such that SUBST(θ, p₁ ∧ … ∧ pₙ) = SUBST(θ, p₁' ∧ … ∧ pₙ')
            for some p₁', …, pₙ' in KB
            q' ← SUBST(θ, q)
            if q' does not unify with some sentence already in KB or new then
                add q' to new
                φ ← UNIFY(q', α)
                if φ is not fail then return φ
        add new to KB
    return false
Exercise

Suppose we have the Ancestry rules and a single fact
mother(sue,fred)

- What, if any, new facts beyond parent/child would be
  inferred?
The inner loop performing pattern-matching in FOL-FC-ASK has exponential complexity. However,

- The number of facts is relatively small in practice
- If the graph of rules is a tree the algorithm is linear
- Heuristics to select the rules exist
- Incremental forward chaining improves performance
Backward Chaining

Backward Chaining uses AND-OR search and lifted modus ponens on KBs of definite clauses. We will look at two variations on the implementation.

- recursive version
- version using *generators*. 
Recursive FOL-BC-ASK

function FOL-BC-ASK(KB, goals, θ) returns a set of substitutions
inputs: KB, a knowledge base
        goals, a list of conjuncts forming a query (θ already applied)
        θ, the current substitution, initially the empty substitution { }
local variables: answers, a set of substitutions, initially empty

if goals is empty then return {θ}
q' ← SUBST(θ, FIRST(goals))
for each sentence r in KB where STANDARDIZE-APART(r) = \( p_1 \land \ldots \land p_n \Rightarrow q \)
             and \( \theta' \leftarrow UNIFY(q, q') \) succeeds
        new_goals ← \{ p_1, \ldots, p_n | REST(goals) \}
        answers ← FOL-BC-ASK(KB, new_goals, COMPOSE(\theta', \theta)) \cup answers
return answers

Figure 9.6 A simple backward-chaining algorithm.
Generators in Python

Example Fibonacci Sequence

def fib():
    Fm1 = 1
    yield Fm1
    Fm2 = 1
    yield Fm2
    while True:
        Fn = Fm1+Fm2
        Fm2 = Fm1
        Fm1 = Fn
        yield Fn
function FOL-BC-ASK(KB, query) returns a generator of substitutions
    return FOL-BC-OR(KB, query, { })

generator FOL-BC-OR(KB, goal, θ) yields a substitution
    for each rule \( \text{lhs} \Rightarrow \text{rhs} \) in FETCH-RULES-FOR-GOAL(KB, goal) do
        \((\text{lhs}, \text{rhs}) \leftarrow \text{STANDARDIZE-VARIABLES}((\text{lhs}, \text{rhs}))\)
        for each \( θ' \) in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
            yield \( θ' \)

generator FOL-BC-AND(KB, goals, θ) yields a substitution
    if \( θ = \text{failure} \) then return
    else if \( \text{LENGTH}(\text{goals}) = 0 \) then yield \( θ \)
    else do
        \( \text{first, rest} \leftarrow \text{FIRST}(\text{goals}), \text{REST}(\text{goals})\)
        for each \( θ' \) in FOL-BC-OR(KB, \text{SUBST}(θ, \text{first}), θ) do
            for each \( θ'' \) in FOL-BC-AND(KB, rest, θ') do
                yield \( θ'' \)
Prolog

Backward Chaining leads to a class of logical programming languages exemplified by *Prolog*.

- may be interpreted or compiled
- uses upper case vars, lower case constants, write conclusion first
- has some non-logical parts, e.g. math
- is not complete, suffers from infinite recursion

Our previous example in Prolog

```prolog
friends(X,Y) :- likes(X,Z), likes(Y,Z).
likes(bill,movies).
likes(sally,movies).
likes(bob,pizza).
```
Next Actions

There is **no class meeting next time** *(3/17)*, however
- Read about Planning, AIAMA 10.1-10.3
- Watch the prerecorded lecture
- There is no warmup

**Reminder:** PS 2 is due 3/23.