Last Time

- First Order Logic
  - Reasons about objects, predicates
  - Introduces equality and quantifiers

- Brief excursion into Prolog
  - To be finished and related to more in depth today
Outline

- Reducing first-order inference to propositional inference
- Unification
- Generalized Modus Ponens
- Forward chaining
- Backward chaining
- Resolution
Universal instantiation (UI)

- Every instantiation of a universally quantified sentence is entailed by it:

\[
\forall \forall \alpha \\
\underline{\text{Subst}\{\nu/g\}, \alpha}
\]

for any variable \(\nu\) and ground term \(g\)

- E.g., \(\forall x \ King(x) \land Greedy(x) \Rightarrow Evil(x)\) yields:
  \[
  \begin{align*}
  King(John) \land Greedy(John) & \Rightarrow Evil(John) \\
  King(Richard) \land Greedy(Richard) & \Rightarrow Evil(Richard) \\
  King(Father(John)) \land Greedy(Father(John)) & \Rightarrow Evil(Father(John))
  \end{align*}
  \]
  ...

Existential instantiation (EI)

For any sentence $\alpha$, variable $v$, and constant symbol $k$ that does not appear elsewhere in the knowledge base:

$$\exists v \alpha \quad \text{Subst}\{\{v/k\}, \alpha\}$$

E.g., $\exists x \ Crown(x) \land OnHead(x, \text{John})$ yields:

$$\text{Crown}(C_1) \land \text{OnHead}(C_1, \text{John})$$

provided $C_1$ is a new constant symbol, called a Skolem constant
Suppose the KB contains just the following:

\[ \forall x \text{ King}(x) \land \text{ Greedy}(x) \Rightarrow \text{ Evil}(x) \]
\[ \text{ King}(\text{John}) \]
\[ \text{ Greedy}(\text{John}) \]
\[ \text{ Brother}(\text{Richard}, \text{John}) \]

- Instantiating the universal sentence in all possible ways, we have:
  \[ \text{ King}(\text{John}) \land \text{ Greedy}(\text{John}) \Rightarrow \text{ Evil}(\text{John}) \]
  \[ \text{ King}(\text{Richard}) \land \text{ Greedy}(\text{Richard}) \Rightarrow \text{ Evil}(\text{Richard}) \]
  \[ \text{ King}(\text{John}) \]
  \[ \text{ Greedy}(\text{John}) \]
  \[ \text{ Brother}(\text{Richard}, \text{John}) \]

- The new KB is **propositionalized**: proposition symbols are
  \[ \text{ King}(\text{John}), \text{ Greedy}(\text{John}), \text{ Evil}(\text{John}), \text{ King}(\text{Richard}), \text{ etc.} \]
Propositionalization

- Every FOL KB can be propositionalized so as to preserve entailment
  - Convert it to propositional logic
  - A ground sentence is entailed by new KB iff entailed by original KB

- Idea: propositionalize KB and query, apply resolution, return result

- But there’s a problem: with function symbols, there are infinitely many ground terms:
Propositionalization, continued

Theorem: Herbrand (1930). If a sentence $\alpha$ is entailed by an FOL KB, it is entailed by a finite subset of the propositionalized KB.

Idea: For $n = 0$ to $\infty$ do
1. create a propositional KB by instantiating with depth-$n$ terms
2. see if $\alpha$ is entailed by this KB

Problem: works if $\alpha$ is entailed, loops if $\alpha$ is not entailed

Theorem: Turing (1936), Church (1936) Entailment for FOL is semidecidable (algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every non-entailed sentence.)
Problems with propositionalization

- Propositionalization seems to generate lots of irrelevant sentences.

  E.g., from:
  \[ \forall x \text{ King}(x) \land \text{ Greedy}(x) \Rightarrow \text{ Evil}(x) \]
  \[ \text{ King}(\text{ John}) \]
  \[ \forall y \text{ Greedy}(y) \]
  \[ \text{ Brother}(\text{ Richard}, \text{ John}) \]

  it seems obvious that \( \text{ Evil}(\text{ John}) \), but propositionalization produces lots of facts such as \( \text{ Greedy}(\text{ Richard}) \) that are irrelevant

- With \( p \) \( k \)-ary predicates and \( n \) constants, there are \( p \cdot n^k \) instantiations.
  - E.g., \( p=1 \) \( k=2 \) \( n=3 \), \( \text{ Rel(_,_,)} \).
    \[ 3 \times 3 = 3^2 = 9 \] possibilities
Unification

- We can get the inference immediately if we can find a substitution $\theta$ such that $\text{King}(x)$ and $\text{Greedy}(x)$ match $\text{King}(\text{John})$ and $\text{Greedy}(y)$

$\theta = \{x/\text{John}, y/\text{John}\}$ works

- Unify($\alpha, \beta$) = $\theta$ if $\alpha\theta = \beta\theta$

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$\theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knows(John, x)</td>
<td>Knows(John, Jane)</td>
<td>θ</td>
</tr>
<tr>
<td>Knows(John, x)</td>
<td>Knows(y, OJ)</td>
<td>θ</td>
</tr>
<tr>
<td>Knows(John, x)</td>
<td>Knows(y, Mother(y))</td>
<td>θ</td>
</tr>
<tr>
<td>Knows(John, x)</td>
<td>Knows(x, OJ)</td>
<td>θ</td>
</tr>
</tbody>
</table>

- **Standardizing apart** eliminates overlap of variables, e.g., Knows($z_{17}$, OJ)
To unify $\text{Knows}(John,x)$ and $\text{Knows}(y,z)$,

$\theta = \{y/John, x/z \}$ or
$\theta = \{y/John, x/John, z/John\}$

The first unifier is **more general** than the second.

There is a single **most general unifier** (MGU) that is unique up to renaming of variables.

$\text{MGU} = \{ y/John, x/z \}$
The unification algorithm

function $\text{UNIFY}(x, y, \theta)$ returns a substitution to make $x$ and $y$ identical

inputs: $x$, a variable, constant, list, or compound
        $y$, a variable, constant, list, or compound
        $\theta$, the substitution built up so far

if $\theta = \text{failure}$ then return failure
else if $x = y$ then return $\theta$
else if $\text{VARIABLE}?(x)$ then return $\text{UNIFY-VAR}(x, y, \theta)$
else if $\text{VARIABLE}?(y)$ then return $\text{UNIFY-VAR}(y, x, \theta)$
else if $\text{COMPOUND}?(x)$ and $\text{COMPOUND}?(y)$ then
    return $\text{UNIFY}($ARGS$[x], $ARGS$[y], \text{UNIFY}($Op$[x], $Op$[y], \theta))$
else if $\text{LIST}?(x)$ and $\text{LIST}?(y)$ then
    return $\text{UNIFY}($REST$[x], $REST$[y], \text{UNIFY}($FIRST$[x], $FIRST$[y], \theta))$
else return failure
The unification algorithm

function UNIFY-VAR(var, x, θ) returns a substitution
inputs: var, a variable
        x, any expression
        θ, the substitution built up so far

if \{var/val\} ∈ θ then return UNIFY(val, x, θ)
else if \{x/val\} ∈ θ then return UNIFY(var, val, θ)
else if OCCUR-CHECK?(var, x) then return failure
else return add \{var/x\} to θ
Let’s do one together

function \textsc{Unify}(x, y, \theta) \text{ returns a substitution to make } x \text{ and } y \text{ identical}
inputs: \ x, \ \text{a variable, constant, list, or compound}
\ y, \ \text{a variable, constant, list, or compound}
\ \theta, \ \text{the substitution built up so far}

if \ \theta = \text{failure} \text{ then return failure}
else if x = y \text{ then return } \theta
else if \ \text{Variable?(x)} \text{ then return } \textsc{Unify-Var}(x, y, \theta)
else if \ \text{Variable?(y)} \text{ then return } \textsc{Unify-Var}(y, x, \theta)
else if \ \text{Compound?(x)} \text{ and } \ \text{Compound?(y)} \text{ then}
\quad \text{return } \textsc{Unify}(\text{Args}[x], \text{Args}[y], \text{Unify}(\text{Op}[x], \text{Op}[y], \theta))
else if \ \text{List?(x)} \text{ and } \ \text{List?(y)} \text{ then}
\quad \text{return } \textsc{Unify}(\text{Rest}[x], \text{Rest}[y], \text{Unify}(\text{First}[x], \text{First}[y], \theta))
else \text{ return failure}

function \textsc{Unify-Var}(\textit{var}, x, \theta) \text{ returns a substitution}
inputs: \ \textit{var}, \ \text{a variable}
\ \ x, \ \text{any expression}
\ \ \ \ \theta, \ \text{the substitution built up so far}

if \ \{\textit{var}/\text{val}\} \in \theta \text{ then return } \textsc{Unify}(\text{val}, x, \theta)
else if \ \{x/\text{val}\} \in \theta \text{ then return } \textsc{Unify}(\textit{var}, \text{val}, \theta)
else if \ \text{OCCUR-CHECK?(\textit{var}, x)} \text{ then return failure}
else \text{ return add } \{\textit{var}/x\} \text{ to } \theta
**Generalized Modus Ponens (GMP)**

\[
p_1', p_2', \ldots, p_n', (p_1 \land p_2 \land \ldots \land p_n \Rightarrow q)
\]

\[
\frac{\text{q}}{q_\theta}
\]

where \( p_i' \theta = p_i \theta \) for all \( i \)

- \( p_1' \) is \( \text{King}(\text{John}) \)
- \( p_1 \) is \( \text{King}(x) \)
- \( p_2' \) is \( \text{Greedy}(y) \)
- \( p_2 \) is \( \text{Greedy}(x) \)
- \( \theta \) is \( \{x/\text{John}, y/\text{John}\} \)
- \( q \) is \( \text{Evil}(x) \)
- \( q_\theta \) is \( \text{Evil}(\text{John}) \)

- GMP used with KB of **definite clauses** (exactly one positive literal)
  - n.b. recall Horn form allows at most one positive literal (less restrictive)

- All variables assumed universally quantified
Let’s do an example with a KB

- The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

- Prove that Colonel West is a criminal
The example KB in FOL

... it is a crime for an American to sell weapons to hostile nations:

\[ \text{American}(x) \land \text{Weapon}(y) \land \text{Sells}(x,y,z) \land \text{Hostile}(z) \Rightarrow \text{Criminal}(x) \]

Nono ... has some missiles, i.e., \( \exists x \text{ Owns}(\text{Nono},x) \land \text{Missile}(x) \):

\[ \text{Owns}(\text{Nono},M_1) \land \text{Missile}(M_1) \]

... all of its missiles were sold to it by Colonel West

\[ \text{Missile}(x) \land \text{Owns}(\text{Nono},x) \Rightarrow \text{Sells}(\text{West},x,\text{Nono}) \]

Missiles are weapons:

\[ \text{Missile}(x) \Rightarrow \text{Weapon}(x) \]

An enemy of America counts as "hostile":

\[ \text{Enemy}(x,\text{America}) \Rightarrow \text{Hostile}(x) \]

West, who is American ...

\[ \text{American}(\text{West}) \]

The country Nono, an enemy of America ...

\[ \text{Enemy}(\text{Nono},\text{America}) \]
Forward chaining algorithm

function FOL-FC-Ask(KB, α) returns a substitution or false

repeat until new is empty
    new ← { }
    for each sentence r in KB do
        (p_1 ∧ ... ∧ p_n ⇒ q) ← STANDARDIZE-APART(r)
        for each θ such that (p_1 ∧ ... ∧ p_n)θ = (p'_1 ∧ ... ∧ p'_n)θ
            for some p'_1, ..., p'_n in KB
                q' ← SUBST(θ, q)
                if q' is not a renaming of a sentence already in KB or new then do
                    add q' to new
                    φ ← UNIFY(q', α)
                    if φ is not fail then return φ
                add new to KB
    return false
Forward chaining proof

\[\text{American(West)} \quad \text{Missile(MI)} \quad \text{Owns(Nono,MI)} \quad \text{Enemy(Nono,America)}\]
Forward chaining proof
Forward chaining proof
Soundness of GMP

- Need to show that

\[ p_1', \ldots, p_n', (p_1 \land \cdots \land p_n \Rightarrow q) \models q_\theta \]

provided that \( p_i'\theta = p_i\theta \) for all \( i \)

- Lemma: For any sentence \( p \), we have \( p \models p_\theta \) by UI

1. \((p_1 \land \cdots \land p_n \Rightarrow q) \models (p_1 \land \cdots \land p_n \Rightarrow q_\theta) = (p_1\theta \land \cdots \land p_n\theta \Rightarrow q_\theta)\)
2. \( p_1', \ldots, p_n' \models p_1' \land \cdots \land p_n' \models p_1'\theta \land \cdots \land p_n'\theta \)
3. From 1 and 2, \( q_\theta \) follows by ordinary Modus Ponens
Properties of forward chaining

- Sound and complete for first-order definite clauses
- **Datalog** = first-order definite clauses + no functions
- FC terminates for Datalog in finite number of iterations
- May not terminate in general if $\alpha$ is not entailed
- This is unavoidable: entailment with definite clauses is again semidecidable
Equivalence to CSPs

- Each conjunct can be viewed as a constraint on a variable.
- Every finite CSP can be expressed as a single definite clause together with some facts.

\[\text{Diff(wa,nt)} \land \text{Diff(wa,sa)} \land \text{Diff(nt,q)} \land \text{Diff(nt,sa)} \land \text{Diff(q,nsw)} \land \text{Diff(q,sa)} \land \text{Diff(nsw,v)} \land \text{Diff(nsw,sa)} \land \text{Diff(v,sa)} \Rightarrow \text{Colorable()}\]

\[
\begin{align*}
\text{Diff(Blue,Red)} & \quad \text{Diff(Blue,Green)} \\
\text{Diff(Green,Red)} & \quad \text{Diff(Green,Blue)} \\
\text{Diff(Red,Blue)} & \quad \text{Diff(Red,Green)}
\end{align*}
\]
Improving efficiency

The algorithm presented earlier isn’t efficient. Let’s make it better.

1. Matching itself is expensive, **Database indexing** allows $O(1)$ retrieval of known facts
   - e.g., query a table where all instantiations of $p(x)$ are stored; $Missile(x)$ retrieves $Missile(M_1)$

For predicates with many subgoals, the conjunct ordering problem applies
   - e.g., for $Missile(x) \land Owns(Nono,x) \Rightarrow Sells(West, x, Nono)|$ if there are many things owned by Nono, perhaps better to start with $Missile(x)$ conjunct
2. Incremental forward chaining: Only match rules on iteration $k$ if a premise was added on iteration $k-1$
   - Original algorithm discards partially matched rules
   - Instead, keep track of conjuncts matched to avoid duplicate work

   
   - Match each rule whose premise contains a newly added positive literal

   Leads to the development of Rete (“Ree-Tee”) networks in real world production systems

3. Irrelevant Facts: several ways to address … let’s segue to Backward Chaining.
Backward chaining algorithm

function FOL-BC-Ask(KB, goals, \( \theta \)) returns a set of substitutions

inputs: KB, a knowledge base

goals, a list of conjuncts forming a query

\( \theta \), the current substitution, initially the empty substitution \{ \}

local variables: ans, a set of substitutions, initially empty

if goals is empty then return \{ \theta \}

\( q' \leftarrow \text{SUBST}(\theta, \text{FIRST(goals)}) \)

for each \( r \) in KB where \text{STANDARDIZE-APART}(r) = ( p_1 \land \ldots \land p_n \Rightarrow q )

and \( \theta' \leftarrow \text{UNIFY}(q, q') \) succeeds

\( ans \leftarrow \text{FOL-BC-Ask}(KB, [p_1, \ldots, p_n|\text{REST(goals)}], \text{COMPOSE(\theta, \theta'))} \cup ans \)

return \( ans \)

\[
\text{SUBST(COMPOSE(}\theta_1, \theta_2), p) = \text{SUBST}(\theta_2, \text{SUBST}(\theta_1, p))
\]
Backward chaining example

\textit{Criminal(West)}
Backward chaining example
Backward chaining example
Backward chaining example
Backward chaining example
Backward chaining example
Backward chaining example
Properties of backward chaining

- Depth-first recursive proof search: space is linear in size of proof
- Incomplete due to infinite loops
  \[ \Rightarrow \text{fix by checking current goal against every goal on stack} \]
- Inefficient due to repeated subgoals (both success and failure)
  \[ \Rightarrow \text{fix using caching of previous results (extra space)} \]
- Widely used for logic programming
Logic programming: Prolog

- Backward chaining with Horn clauses + bells & whistles

- Program = set of clauses = head :- literal₁, … literalₙ.
  
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).

- Depth-first, left-to-right backward chaining
- Built-in predicates for arithmetic etc., e.g., X is Y*Z+3
- Built-in predicates that can have side effects (e.g., input and output predicates, assert/retract predicates)
- Closed-world assumption / database semantics ("negation as failure")
  - e.g., given alive(X) :- not dead(X).
  - alive(joe) succeeds if dead(joe) fails

- No checks for infinite recursion
- No occurs check for unification
Resolution: recap and look at FOL

● Full first-order version:

\[
\begin{align*}
\ell_1 \lor \cdots \lor \ell_k, & \quad m_1 \lor \cdots \lor m_n \\
(\ell_1 \lor \cdots \lor \ell_{i-1} \lor \ell_{i+1} \lor \cdots \lor \ell_k \lor m_1 \lor \cdots \lor m_{j-1} \lor m_{j+1} \lor \cdots \lor m_n)\theta
\end{align*}
\]

where \(\text{Unify}(\ell_i, \neg m_j) = \theta\).

● The two clauses are assumed to be standardized apart so that they share no variables.

● For example,

\[
\begin{align*}
\neg \text{Rich}(x) \lor \text{Unhappy}(x), & \quad \text{Rich}(\text{Ken}) \\
\text{Unhappy}(\text{Ken})
\end{align*}
\]

with \(\theta = \{x/\text{Ken}\}\)

● Apply resolution steps to \(\text{CNF}(\text{KB} \land \neg \alpha)\); complete for FOL*
Conversion to CNF

1. Everyone who loves all animals is loved by someone:
   \[ \forall x \left[ \forall y \ Animal(y) \Rightarrow Loves(x,y) \right] \Rightarrow \left[ \exists y \ Loves(y,x) \right] \]

2. Eliminate biconditionals and implications
   \[ \forall x \left[ \neg \forall y \neg Animal(y) \vee Loves(x,y) \right] \vee \left[ \exists y \ Loves(y,x) \right] \]

3. Move \( \neg \) inwards:
   \[ \neg \forall x \ p \equiv \exists x \ \neg p, \]
   \[ \neg \exists x \ p \equiv \forall x \ \neg p \]
   \[ \forall x \left[ \exists y \ \neg \left( \neg Animal(y) \vee Loves(x,y) \right) \right] \vee \left[ \exists y \ Loves(y,x) \right] \]
   \[ \forall x \left[ \exists y \ \neg Animal(y) \wedge \neg Loves(x,y) \right] \vee \left[ \exists y \ Loves(y,x) \right] \]
   \[ \forall x \left[ \exists y \ Animal(y) \wedge \neg Loves(x,y) \right] \vee \left[ \exists y \ Loves(y,x) \right] \]
3. Standardize variables: each quantifier should use a different one
   \[ \forall x \left[ \exists y \ Animal(y) \land \neg Loves(x,y) \right] \lor \left[ \exists z \ Loves(z,x) \right] \]

4. Skolemize: a more general form of existential instantiation.
   Each existential variable is replaced by a Skolem function of the enclosing
   universally quantified variables:
   \[ \forall x \left[ Animal(F(x)) \land \neg Loves(x,F(x)) \right] \lor Loves(G(x),x) \]
   
   Why do we need a function and not a variable?

5. Drop universal quantifiers:
   \[ [Animal(F(x)) \land \neg Loves(x,F(x))] \lor Loves(G(x),x) \]

6. Distribute \( \lor \) over \( \land \):
   \[ [Animal(F(x)) \lor Loves(G(x),x)] \land [\neg Loves(x,F(x)) \lor Loves(G(x),x)] \]
Resolution proof: definite clauses

\( \neg American(x) \lor \neg Weapon(y) \lor \neg Sells(x,y,z) \lor \neg Hostile(z) \lor Criminal(x) \)

\( \neg Criminal(West) \)
Resolution proof: definite clauses

\[ \neg \text{American}(x) \lor \neg \text{Weapon}(y) \lor \neg \text{Sells}(x,y,z) \lor \neg \text{Hostile}(z) \lor \text{Criminal}(x) \]
Resolution proof: definite clauses

\[ \neg \text{American}(x) \lor \neg \text{Weapon}(y) \lor \neg \text{Sells}(x,y,z) \lor \neg \text{Hostile}(z) \lor \text{Criminal}(x) \]
Resolution proof: definite clauses

\[ \neg \text{American}(x) \lor \neg \text{Weapon}(y) \lor \neg \text{Sells}(x,y,z) \lor \neg \text{Hostile}(z) \lor \text{Criminal}(x) \]

\[ \neg \text{Criminal}(\text{West}) \]

\[ \neg \text{American}(\text{West}) \lor \neg \text{Weapon}(y) \lor \neg \text{Sells}(\text{West},y,z) \lor \neg \text{Hostile}(z) \]

\[ \neg \text{Missile}(x) \lor \text{Weapon}(x) \]

\[ \neg \text{Missile}(\text{M1}) \]

\[ \neg \text{Missile}(x) \lor \neg \text{Owns}(\text{None},x) \lor \text{Sells}(\text{West},x,\text{None}) \]

\[ \neg \text{Sells}(\text{West},\text{M1},z) \lor \neg \text{Hostile}(z) \]

\[ \neg \text{Missile}(\text{M1}) \lor \neg \text{Owns}(\text{None},\text{M1}) \lor \neg \text{Hostile}(\text{None}) \]

\[ \neg \text{Owns}(\text{None},\text{M1}) \lor \neg \text{Hostile}(\text{None}) \]

\[ \neg \text{Enemy}(x,\text{America}) \lor \text{Hostile}(x) \]

\[ \neg \text{Hostile}(\text{None}) \]

\[ \neg \text{Enemy}(\text{None},\text{America}) \]
Resolution proof: definite clauses

\[ \neg \text{American}(x) \lor \neg \text{Weapon}(y) \lor \neg \text{Sells}(x,y,z) \lor \neg \text{Hostile}(z) \lor \text{Criminal}(x) \]

\[ \neg \text{American}(\text{West}) \]

\[ \neg \text{Missile}(x) \lor \text{Weapon}(x) \]

\[ \neg \text{Missile}(\text{M1}) \]

\[ \neg \text{Missile}(x) \lor \neg \text{Owns}(\text{Nono},x) \lor \text{Sells}(\text{West},x,\text{Nono}) \]

\[ \neg \text{Owns}(\text{Nono},\text{M1}) \]

\[ \neg \text{Enemy}(x,\text{America}) \lor \text{Hostile}(x) \]

\[ \neg \text{Enemy}(\text{Nono},\text{America}) \]
Refutation completeness

- **Resolution** can say *yes* to any entailed sentence but cannot be used to *generate* all entailed sentences
  - E.g., won’t generate \( \text{Animal}(x) \lor \neg \text{Animal}(x) \)
Resolution special cases

- **Factoring**: may need to remove redundant literals (literals that are unifiable)

  - $L(x) \lor G(a,b)$
  - $\neg L(x) \lor G(K,L)$

- To handle equality $x=y$, need to use **demodulation** (sub $x$ for $y$ in some clause that has $x$).

  - $B = \text{Son}(A)$
  - Property($B$)
Summary

- Examined our three strategies for logic inference in FOL:
  - Forward Chaining
  - Backward Chaining (what Prolog uses)
  - Resolution

- To think about: when is each of the three systems the most appropriate?