Extra formulas in UI

- ∀x (King(x) □ Greedy(x)) ⇒ Evil(x)
- King(John)
- Greedy(John)

- Universal instantiation produces unnecessary instantiations:
  - To check KB ⊨ Evil(John)
    - We only need to instantiate { x/John }
    - No other instantiation is needed
  - We will then obtain Evil(John)
    By repeated application of Modus Ponens

Extra Formulas in UI

- ∀x (King(x) □ Greedy(x)) ⇒ Evil(x)
- King(John)
- ∀y Greedy(y)

- To check KB ⊨ Evil(John)
  - We only need to instantiate { x/John, y/John }
    - No other instantiation is needed

- King(John) Greedy(y) (King(x) □ Greedy(x)) ⇒ Evil(x)

Evil(John)

Generalized Modus Ponens

- Let P_1 P_2 … P_n and P'_1 P'_2…. P'_n and Q be atomic formulas whose variables are universally quantified
- Let θ be a substitution such that for all i, $SUBST\{\theta, P'_i\} = SUBST\{\theta, P_i\}$ where θ is a substitution

- Generalized Modus Ponens Rule:

$$\frac{p'_1 \; p'_2 \; \ldots\ldots \; p'_n \qquad p_1 \wedge p_2 \wedge \ldots p_n \Rightarrow q}{SUBST(\theta, q)}$$

- Example:

$$\frac{King(John) \quad Greedy(y) \quad (King(x) \; \square \; Greedy(x)) \Rightarrow Evil(x)}{Evil(John)}$$

  - Here, $\theta = \{ x / John, \; y / John \}$

- Generalized Modus Ponens is Sound

Generalized Modus Ponens

- Generalized Modus Ponens lifts Modus Ponens applied to ground terms to quantified variables.

- We looked at three algorithms for Entailment in Propositional Logic
  - Forward Chaining
  - Backward Chaining
  - Resolution Algorithm

- We will look at how these can be generalized to First-Order Logic

- Before that we will look at Unification
  - Substitutions that make two atomic sentences look the same

Unification

- To lift the inferences, we first need to find a substitution that produces identical formulas:
  - UNIFY(P,Q) returns θ such that SUBST(θ, P) = SUBTS(θ, Q)
    - If such a θ exists

- Examples:
  - UNIFY(Knows(John, x), Knows(John, Jane))
    - { x/Jane }
  - UNIFY(Knows(John, x), Knows(y, Bill))
    - {x/Bill, y/John }
  - UNIFY(Knows(John, x), Knows(y, Mother(y)))
    - { y/John, x/Mother(John) }
  - UNIFY(Knows(John, x), Knows(x, Elizabeth))
    - Failure

Unification

- UNIFY(Knows(John, x), Knows(x, Elizabeth))
  - Failure

- Here, the problem was that the same variable was used in both sentences
  - Solution : Use different quantified variables in each statement

- UNIFY(Knows(John, x), Knows(z, Elizabeth))
  - { z/John, x/Elizabeth }

Unification

- UNIFY(Knows(John, x), Knows(y, z))

- What is the unifier here?
  - { x/John, y/John, z/John }
  - { y/ John, z/ x }

- { y/ John, z/ x } is More General than { x/John, y/John, z/John }
  - Because we can obtain { x/John, y/John, z/John } by one more step { x/John }

- Most General Unifier: All other substitutions can be obtained from this
  - Always exists if the pair is unifiable
  - Unique up to renaming and substitution

Algorithm to obtain Most General Unifier

- FindMGU(P, Q) : Takes two atomic formulas and returns MGU
  - If P and Q are atoms from Different predicates then RETURN Failure
  - Else P = R($t_1 t_2 \ldots\ldots t_n$) and Q = R($t'_1 t'_2 \ldots\ldots t'_n$)
    - Return FindSub ([$t_1 t_2 \ldots\ldots t_n$], [$t'_1 t'_2 \ldots\ldots t'_n$] , { })

- FindSub (u, v, θ)
  - If u and v are lists of size 1
    - If u and v are ground terms
      - If $u \neq v$ then RETURN Failure
      - If u = v then Return θ
    - If u is a variable then call VarUnify(u, v, θ)
      - If θ is Failure then Return Failure
      - Rewrite u and v with the new θ
    - If v is a variable then call VarUnify(v, u, θ)
      - If θ is Failure then Return Failure
      - Rewrite u and v with the new θ
    - If u = f($t_1 t_2 \ldots\ldots t_n$) and v = g($t'_1 t'_2 \ldots\ldots t'_m$) then RETURN Failure
    - If u = f($t_1 t_2 \ldots\ldots t_n$) and v = f($t'_1 t'_2 \ldots\ldots t'_n$) then
      RETURN FindSub ([$t_1 t_2 \ldots\ldots t_n$], [$t'_1 t'_2 \ldots\ldots t'_n$] , θ)
  - For every i = 1 to |u|
    - θ = FindSub (t_i , t'_i, θ)
    - If θ is Failure then Return Failure
    - Rewrite u and v with the new θ
  - Return θ

- VarUnify(x, u, θ) :
  - If x = u then Return θ
  - Else If x occurs in u then RETURN Failure
  - Else If { x / t} is already in θ then
    RETURN FindSub(t, u, θ)
  - Else Return (θ U { x / u })

- Some examples to try out:
  - P(x, f(g(z))) P(f(z), x)
  - P(A,A,B) , P(x,y,z)
  - Q(y,G(A,B)), Q(G(x,x),y)
  - Q(y,G(A,A)), Q(G(x,x),y)
  - Older(Father(y),y), Older(Father(x),Jerry)
  - Knows(Father(y),y), Knows(x,x)

- Checking whether x occurs in u makes the entire algorithm quadratic
- There are clever ways to have a linear time algorithm to find MGU

Storage and Retreival of KB

- Store(P) stores P in the KB [More general than TELL]
- Fetch(P) returns all unifiers with some sentence in the KB
  [More general than ASK and ASKVars]

- Store the KB as a single List
  - Leads to inefficient unification

- How can we store the KB so that Feth(P) can be answered efficiently?

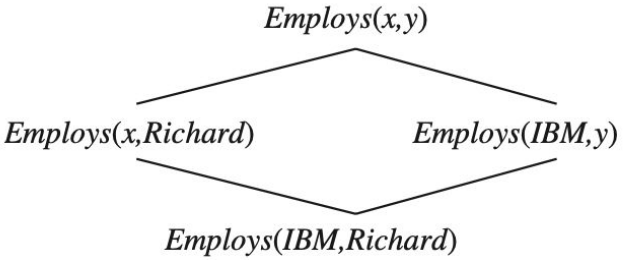
- Predicate Indexing : Stores each predicate in a different bucket
  - Works well if there are many predicates, each with few clauses
  - If there are few predicates but each has a lot of clauses :
    - Leads to inefficient unification
      - Knows (x,y) and Knows(x, Richard) should scan the entire bucket

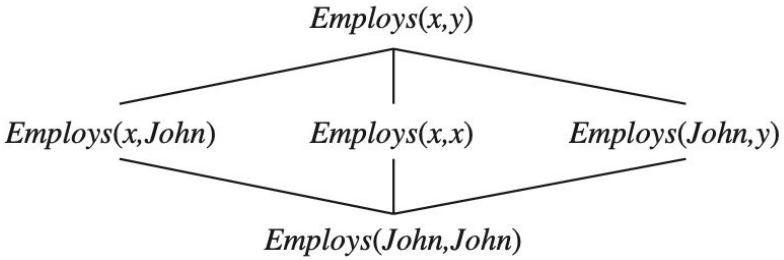
  - One solution : Hash table with second argument for each bucket
    - Knows(Richard, y) will need Hash table for first argument

- Typically stored with multiple index keys (Like indexing in Database records)

Storage and Retreival of KB

- When we add something new to KB can we store all possible queries that unifies with it as a subsumption lattice.





- The child node is obtained by a single substitution of its parent
- The highest common descendant is the most general unifier
- For n arguments, the lattice will have $O(2^n)$ values in the lattice
  - Good for predicates with small number of arguments

- Should we store the subsumption lattice or not is an engineering decision

Forward Chaining in First Order Logic

- First-Order definite Clause has exactly one positive literal
  - Can be written as an implication where antecedent is a conjunction of positive literals and consequent is a positive literal
    - King(x) □ Greedy(x)) ⇒ Evil(x)
    - King(John) is also a definite clause
  - No existential variables are allowed
    - All variables that occur are implicitly assumed to be universally quantified

Forward Chaining in First Order Logic : Example

- 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.

- American(x) Λ Weapon(y) Λ Sells(x, y, z) Λ Hostile(z) ⇒ Criminal(x)
- Owns(Nono, M1)
- Missile(M1)
- Owns(Nono, x) Λ Missile(x)⇒ Sells(West, x, Nono)
- Missile(x) ⇒ Weapon(x)
- Enemy(x, America) ⇒ Hostile(x)
- American(West)
- Enemy(Nono, America)

- This KB is in Datalog
  - Definite clauses with no function symbols

Forward Chaining in First Order Logic

- (American(x) ∧ Weapon(y) ∧
  Sells(x, y, z) ∧ Hostile(z)) ⇒
  Criminal(x)
- Owns(Nono, M1)
- Missile(M1)
- (Owns(Nono, x) ∧ Missile(x)) ⇒
  Sells(West, x, Nono)
- Missile(x) ⇒ Weapon(x)
- Enemy(x, America) ⇒ Hostile(x)
- American(West)
- Enemy(Nono, America)

- With {x/M1} add Sells(West,M1,Nono)
- With {x/M1} add Weapon{M1}
- With {x/Nono} add Hostile(Nono)
- With {x/West, y/M1, z/Nono) add Criminal(West)

Forward Chaining in First Order Logic

- Everyone likes Desserts. Everyone who lives inside IITM likes ice creams. All ice-creams are desserts. Everyone lives inside IITM and Anantha is one of them.

- Dessert(x) ⇒ Likes(y, x)
- Lives(x, IITM) ⇒ Likes(x, Ice-cream)
- Dessert(Ice-cream)
- Lives(x, IITM)
- Lives(Anantha, IITM)

- With { } add Likes(x, Ice-cream)
- With {x/Ice-cream}
  add Likes(y, Ice-cream) (?)
- In first step, with {x/Anantha} we get Likes(Anantha, Ice-cream)

Forward Chaining in First Order Logic

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

  while *true* **do**
    new $\leftarrow \{\}$ // *The set of new sentences inferred on each iteration*
    for each *rule* **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(*rule*)
      for each θ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
          for some $p'_1, \ldots, p'_n$ in KB
        $q' \leftarrow$ SUBST(θ, q)
        if q' does not unify with some sentence already in KB or *new* **then**
          add q' to *new*
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *failure* **then return** ϕ
    if *new* $= \{\}$ **then return** *false*
    add *new* to KB

- The algorithm is Sound
  - Repeated application of Generalized Modus Ponens

Forward Chaining in First Order Logic

```
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

  while true do
    new ← { }        // The set of new sentences inferred on each iteration
    for each rule in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
      for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                  for some p'1, ..., p'n 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 failure then return φ
    if new = { } then return false
    add new to KB
```

- Above algorithm is Complete
  - If there are no functions then then we have a bounded number of ground facts
    - Hence the number of iterations is bounded
- If we also have function symbols:
  - No instances can go on an infinite loop
  - Example:
    NatNum(0)
    Natnum(x) ⇒ NatNum(S(x))
- Algorithm Keeps on Adding
  { Natnum(0), Natnum(S(0)),
  Natnum(S(S(0)))..... }

Forward Chaining in First Order Logic

```
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

  while true do
    new ← { }        // The set of new sentences inferred on each iteration
    for each rule in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
      for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                  for some p'1, ..., p'n 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 failure then return φ
    if new = { } then return false
    add new to KB
```

- This algorithm is not efficient
  - Matching rules against facts takes time
  - Algorithm rechecks every rule in every iteration
  - Generates facts not relevant to the goal

Forward Chaining in First Order Logic

- Matching rules against facts takes time
  - Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
    - Should we first find all facts that match Missle(x) or Owns(Nono, x)
    - Conjunct Ordering problem :
      - NP hard to pick optimal ordering
      - Minimum Remaining value Heuristics : Pick whichever has lesser number of remaining facts
- Inner loop is already NP-hard
  - Data complexity is polynomial (only count ground facts, not size of the rule)
  - Has connections to Constraint Satisfaction Problem (CSP)
    - Pattern matching can be encoded as a CSP

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

  while *true* **do**
    $new \leftarrow \{\}$ // *The set of new sentences inferred on each iteration*
    for each *rule* **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(*rule*)
      for each θ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
          for some $p'_1, \ldots, p'_n$ in KB
        $q' \leftarrow$ SUBST(θ, q)
        if q' does not unify with some sentence already in KB or new **then**
          add q' to new
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *failure* **then return** ϕ
    if $new = \{\}$ **then return** *false*
    add new to KB

Forward Chaining in First Order Logic

- Algorithm rechecks every rule in every iteration
  - How to ensure we do not keep deriving things that are already inferred?
- Every new fact derived at step t must use at least one fact derived in step t-1
  - Incremental Forward Chaining

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

  while *true* **do**
    $new \leftarrow \{\}$ // *The set of new sentences inferred on each iteration*
    for each *rule* **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(*rule*)
      for each θ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
          for some $p'_1, \ldots, p'_n$ in KB
        $q' \leftarrow$ SUBST(θ, q)
        if q' does not unify with some sentence already in KB or new **then**
          add q' to new
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *failure* **then return** ϕ
    if $new = \{\}$ **then return** *false*
    add new to KB

Forward Chaining in First Order Logic

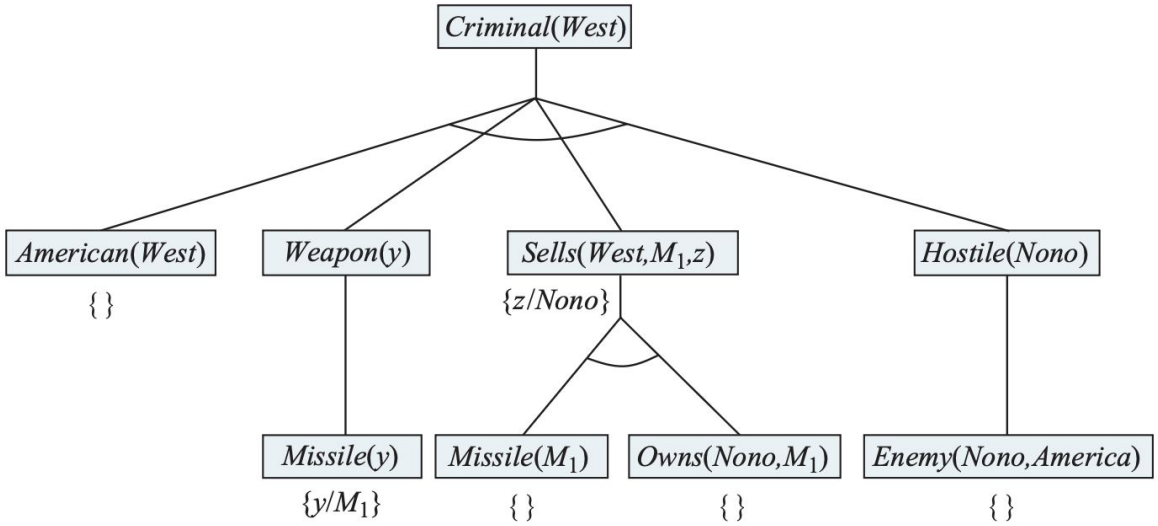
- Generates facts not relevant to the goal
  - Use Backward Chaining
  - Restrict Forward Chaining to a subset of Rules
  - Deductive Databases
    - Like Relational DBs but Forward Chaining is built-in.
    - (American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z)) ⇒ Criminal(x)
    - (Magic(x) ∧ American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z)) ⇒ Criminal(x)
    - To check for Criminal(West), add Magic(West) to the KB

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

  while *true* **do**
    $new \leftarrow \{\}$ // *The set of new sentences inferred on each iteration*
    for each *rule* **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(*rule*)
      for each θ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
          for some $p'_1, \ldots, p'_n$ in KB
        $q' \leftarrow$ SUBST(θ, q)
        if q' does not unify with some sentence already in KB or new **then**
          add q' to new
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *failure* **then return** ϕ
    if $new = \{\}$ **then return** *false*
    add new to KB

Backward Chaining in First Order Logic : Example

- (American(x) Λ Weapon(y) Λ Sells(x, y, z) Λ Hostile(z)) ⇒ Criminal(x)
- Owns(Nono, M1)
- Missile(M1)
- (Owns(Nono, x) Λ Missile(x)) ⇒ Sells(West,x,Nono)
- Missile(x) ⇒ Weapon(x)
- Enemy(x, America) ⇒ Hostile(x)
- American(West)
- Enemy(Nono, America)



Should be careful about infinite loops !

Logic Programming

- There are Programming Language to specify KB of definite clauses and make inferences
- Also called Theorem provers
  - Most common : Prolog, Isabelle, Coq, Lean, ...

- Syntax of Prolog:
  - `criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).`
  - `king(john).`
  - `likes(john, alice).`

  - Variables start with capital letters

  - Queries start with `?-`
    - `?-evil(john).`

  - Has builtin arithmetic predicates
    - `tallerThan(X,Y) :- height(X,A), height(Y,B), A > B.`

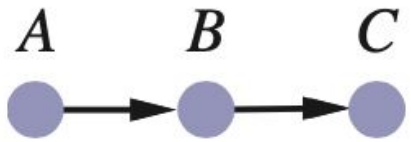
Prolog Backend

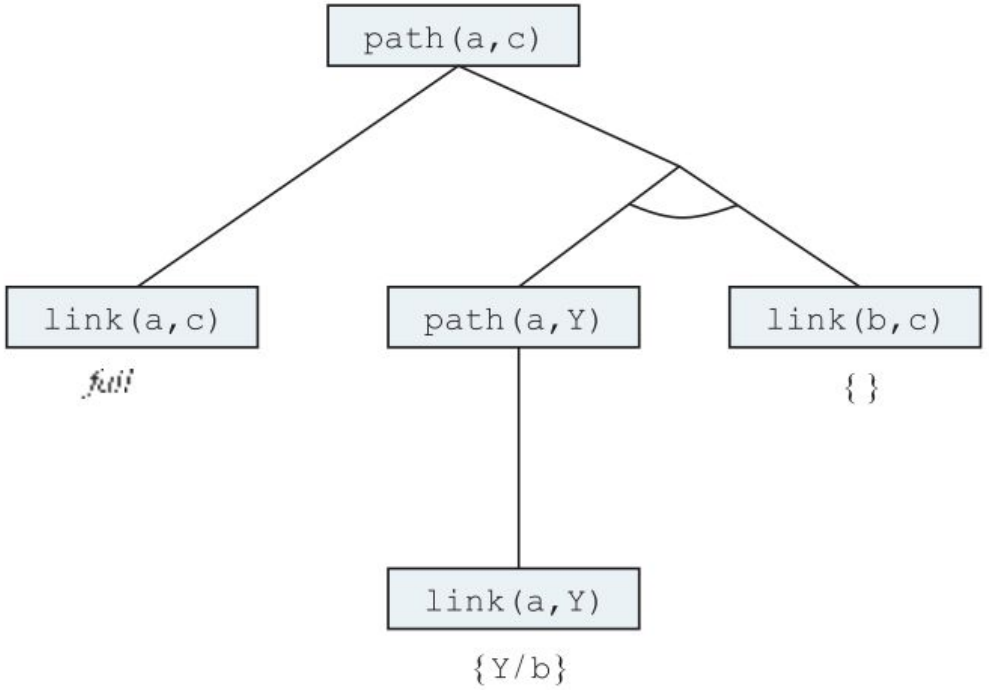
- OccurCheck is omitted from the Prolog Unification Algorithm
  - Onus is on the programmer

- Uses depth-first backward-chaining without checking for infinite loops
  - Fast when used properly
  - Might get into infinite loop even if the logic is correct

Infinite Loops in Prolog

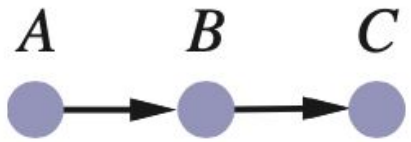
- `path(X,Z) :- link(X,Z).`
- `path(X,Z) :- path(X,Y), link(Y,Z).`
- `link(a,b).`
- `link(b,c).`
- `?-path(a,c).`

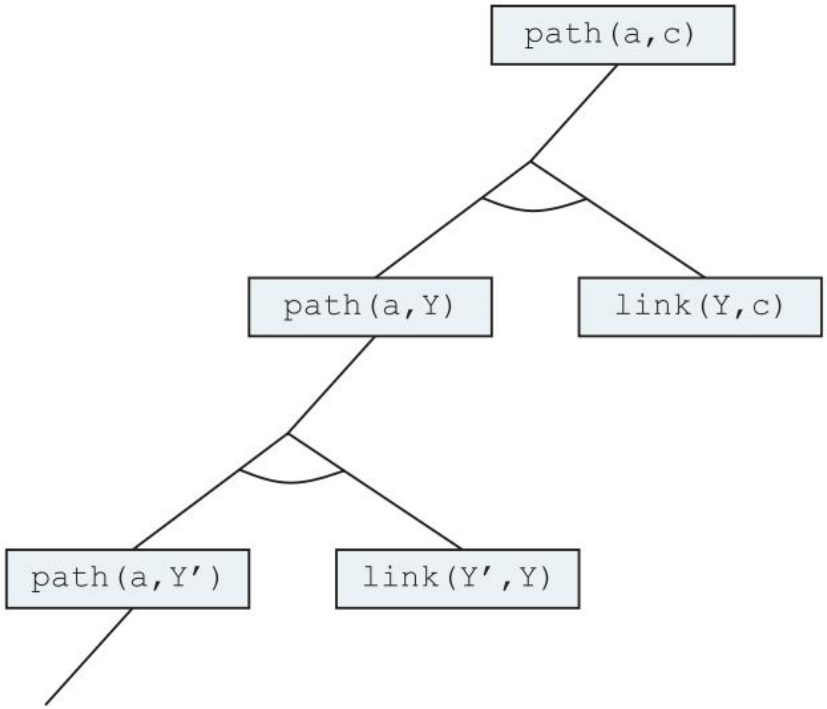




Infinite Loops in Prolog

- `path(X,Z) :- path(X,Y), link(Y,Z).`
- `path(X,Z) :- link(X,Z).`
- `link(a,b).`
- `link(b,c).`
- `?-path(a,c).`

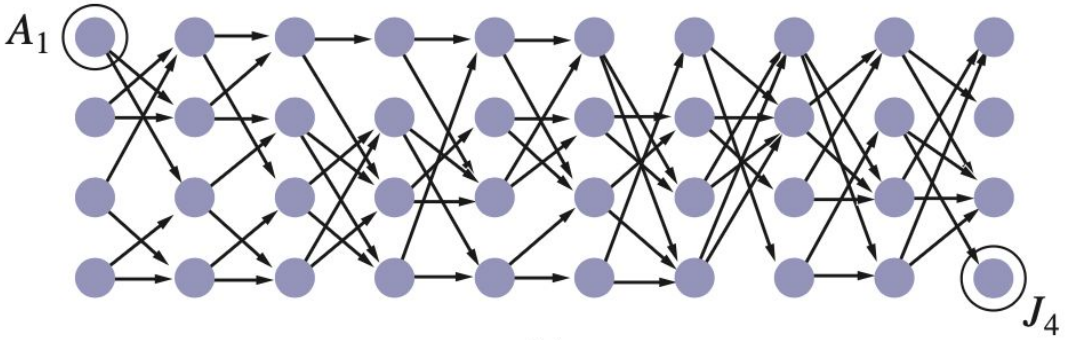




- Prolog is an incomplete theorem prover.

Forward and Backward Chaining computation in Prolog

- `path(X,Z) :- link(X,Z).`
- `path(X,Z) :- path(X,Y), link(Y,Z).`



- `?-path(a1,j4).`
  - Backward Chaining takes 877 inferences
  - Forward Chaining takes 62 inferences
    - It is like Dynamic programming
  - Tabled logic programming tries to avoid exponential blowup in the backtracking

Database semantics in Prolog

- Prolog uses Database semantics
  - Every constant and ground term refers to distinct object
  - Only sentences that are true are those entailed by the KB
  - There are no other domain elements except for the constants mentioned

- Weaker than First Order Logic

- If you can model your KB in Prolog (THEN DO IT!)
  - No need to use the First Order Logic theorem prover

Constraint Logic Programming

- Till now we assumed that the domain is finite
- What if we want to find solutions over infinite domains?
  - Natural Numbers
- Example:
  - `triangle(X,Y,Z) :- X>0, Y>0, Z>0, X+Y>Z, Y+Z>X, X+Z>Y.`
  - ASKVars(KB, triangle(3,4,z))
- Prolog cannot do this because there are infinitely many objects
  - ASK(KB, triangle(3,4,5)) is OK.

Constraint Logic Programming

- Constraint Logic Program outputs constraints on the variables

- Example:
  - `triangle(X,Y,Z) :- X>0, Y>0, Z>0, X+Y>Z, Y+Z>X, X+Z>Y.`
  - ASKVars(KB, triangle(3,4,z))

  - Output would be:
    - 1 < z < 7

- Prolog Type Inferences are special case of Constraint Logic Program
  - Where output will always be some equality

Resolution in First Order Logic

- We first need to write the formulas in CNF
  - Variables occurring in the CNF should be universally quantified

- Example: Everyone who loves all animals is loved by someone
  - $\forall x\,(\forall y\,\text{Animal}(y) \Rightarrow \text{Loves}(x,y)) \Rightarrow (\exists y\,\text{Loves}(y,x))$
  - [Eliminate $\Rightarrow$] $\forall x\,\neg(\forall y\,\neg\text{Animal}(y) \vee \text{Loves}(x,y)) \vee (\exists y\,\text{Loves}(y,x))$
  - [Move $\neg$ inside] $\forall x\,(\exists y\,\text{Animal}(y) \wedge \neg\text{Loves}(x,y)) \vee (\exists y\,\text{Loves}(y,x))$
  - [Standardize variables] $\forall x\,(\exists y\,\text{Animal}(y) \wedge \neg\text{Loves}(x,y)) \vee (\exists z\,\text{Loves}(z,x))$

  - [Skolemize Existential Variables]
    $\forall x\,(\text{Animal}(F(x)) \wedge \neg\text{Loves}(x,F(x))) \vee \text{Loves}(G(x),x))$

  - [Drop universal quantifiers] $(\text{Animal}(F(x)) \wedge \neg\text{Loves}(x, F(x))) \vee \text{Loves}(G(x),x))$
  - [Distribute $\wedge$ over $\vee$]
    $(\text{Animal}(F(x)) \vee \text{Loves}(G(x), x)) \wedge (\neg\text{Loves}(x, F(x)) \vee \text{Loves}(G(x),x))$

Resolution Rule for First Order Logic

- $\ell_1 \vee \ell_2 \vee \ldots \ell_{i-1} \vee \ell_i \vee \ell_{i+1} \vee \ldots \ell_k \; m_1 \vee m_2 \vee \ldots m_{j-1} \vee m_j \vee m_{j+1} \vee \ldots m_n$

  ---------- (Resolution Rule)

  $\text{SUBST}(\theta, \; \ell_1 \vee \ldots \ell_{i-1} \vee \ell_{i+1} \vee \ldots \ell_k \vee m_1 \vee \ldots m_{j-1} \vee m_{j+1} \vee \ldots m_n)$

- Where, $\theta = \text{Unify}(\ell_i, m_j)$

- Example:

  $\text{Animal}(F(x)) \vee \text{Loves}(G(x), x) \qquad \neg\text{Loves}(u,v) \vee \text{Feeds}(u,v)$

  $\text{Animal}(F(x)) \vee \text{Feeds}(G(x), x)$

  Where, $\{u/G(x), \; v/x\} = \text{Unify}(\text{Loves}(G(x), x), \neg\text{Loves}(u,v))$

- This rule is called Binary Resolution Rule since it resolves two literals

Resolution Rule for First Order Logic

- Binary Resolution Rule (BRR) itself is not complete for First Order Logic
  - We also need Factoring
- Factoring Rule (FR): Removing Unifiable literals

$$\frac{l_1 \vee l_2}{l_2} \quad \text{(Factoring Rule)}$$ Where l_1 can be unified with l_2

- Example:

$$\frac{P(x) \vee P(G(u))}{P(G(u))}$$

- BRR + FR is complete for First Order Logic
  - KB $\vDash \alpha$ iff (KB $\wedge \neg\alpha$) is unsatisfiable iff empty clause can be derived for (KB $\wedge \neg\alpha$) using BRR+FR

Example Resolution Proofs

- (American(x) Λ Weapon(y) Λ Sells(x, y, z) Λ Hostile(z)) ⇒ Criminal(x)
- Owns(Nono, M1)
- Missile(M1)
- (Owns(Nono, x) Λ Missile(x)) ⇒ Sells(West,x,Nono)
- Missile(x) ⇒ Weapon(x)
- Enemy(x, America) ⇒ Hostile(x)
- American(West)
- Enemy(Nono, America)