# Inference in First Order Logic

Chapter 9

Some material adopted from notes
by Tim Finin,
Andreas Geyer-Schulz,
and Chuck Dyer

# Inference Rules for FOL

- Inference rules for PL apply to FOL as well (Modus Ponens, And-Introduction, And-Elimination, etc.)
- New (sound) inference rules for use with **quantifiers**:
  - Universal Elimination
  - Existential Introduction
  - Existential Elimination
  - Generalized Modus Ponens (GMP)
- **Resolution**
  - Clause form (CNF in FOL)
  - Unification (consistent variable substitution)
  - Refutation resolution (proof by contradiction)

## Universal Elimination $(\forall x)$ P(x) |-- P(c).

- If $(\forall x)$ P(x) is true, then P(c) is true for **any** constant c in the domain of x, i.e.,, $(\forall x)$ P(x) |= P(c).
- Replace all occurrences of x in the scope of $\forall x$ by the **same** ground term (a constant or a ground function).
- Example: $(\forall x)$ eats(Ziggy, x) |-- eats(Ziggy, IceCream)

## Existential Introduction P(c) |-- $(\exists x)$ P(x)

- If P(c) is true, so is $(\exists x)$ P(x), i.e., P(c) |= $(\exists x)$ P(x)
- Replace all instances of the given constant symbol by the same **new** variable symbol.
- Example eats(Ziggy, IceCream) |-- $(\exists x)$ eats(Ziggy, x)

## Existential Elimination

- From $(\exists x)$ P(x) infer P(c), i.e., $(\exists x)$ P(x) |= P(c), where c is a new constant symbol,
  - All we know is there must be some constant that makes this true, so we can introduce a brand new one to stand in for that constant, *even though we don't know exactly what that constant refer to.*
  - Example: $(\exists x)$ eats(Ziggy, x) |= eats(Ziggy, Stuff)

- Things become more complicated when there are universal quantifiers

  $(\forall x)(\exists y)$ eats(x, y) |= $(\forall x)$eats(x, Stuff) ???

  $(\forall x)(\exists y)$ eats(x, y) |= eats(Ziggy, Stuff) ???

  – Introduce a **new** function *food_sk*(x) to stand for $\exists y$ because that y depends on x

  $(\forall x)(\exists y)$ eats(x, y) |-- $(\forall x)$eats(x, food_sk(x))

  $(\forall x)(\exists y)$ eats(x, y) |-- eats(Ziggy, food_sk(Ziggy))

  – What exactly the function food_sk(.) does is unknown, except that it takes x as its argument

- The process of existential elimination is called "*Skolemization*", and the new, unique constants (e.g., Stuff) and functions (e.g., food_sk(.)) are called skolem constants and skolem functions

## Generalized Modus Ponens (GMP)

- Combines And-Introduction, Universal-Elimination, and Modus Ponens
- Ex: P(c), Q(c), $(\forall x)(P(x) \wedge Q(x)) \Rightarrow R(x)$ |-- R(c)

  P(c), Q(c) |-- P(c) ^ Q(c) (*by and-introduction*)

  (∀x)(P(x) ^ Q(x)) => R(x)

  |-- (P(c) ^ Q(c)) => R(c) (*by universal-elimination*)

  P(c) ^ Q(c), (P(c) ^ Q(c)) => R(c) |-- R(c) (*by modus ponens*)
- All occurrences of a quantified variable must be instantiated to the same constant.

  P(a), Q(c), (∀x)(P(x) ^ Q(x)) => R(x) |-- R(c)

  because all occurrences of x must either instantiated to a or c which makes the modus ponens rule not applicable.

# Resolution for FOL

- Resolution rule operates on two *clauses*
  - A clause is a **disjunction** of literals (without explicit quantifiers)
  - Relationship between clauses in KB is **conjunction**
- Resolution Rule for FOL:
  - clause C1: (l_1, l_2, ... ***l_i***, ... l_n) and

    clause C2: (l'_1, l'_2, ... ***l'_j***, ... l'_m)
  - if l_i and l'_j are two **opposite literals** (e.g., P and ~P) and their argument lists can be be made the same (**unified**) by a set of variable bindings θ = {x1/y1, ... Xk/yk} where x1, ... Xk are variables and y1, ... Yk are terms, then derive a new clause (called resolvent)

    subst((l_1, l_2, ... l_n, l'_1, l'_2, ... l'_m), θ)

    where function subst(expression, θ) returns a new expression by applying all variable bindings in θ to the original expression

# We need answers to the following questions

- How to convert FOL sentences to clause form (especially how to remove quantifiers)
- How to unify two argument lists, i.e., how to find their most general unifier (**mgu**) $\theta$
- How to determine which two clauses in KB should be resolved next (among all resolvable pairs of clauses) and how to determine a proof is completed

# Converting FOL sentences to clause form

- *Clauses* are quantifier free CNF of FOL sentences
- Basic ideas
  - How to handle quantifiers
    - Careful on quantifiers with preceding negations (explicit or implicit)

      $\sim\forall x\ P(x)$ is really $\exists x\ \sim P(x)$

      $(\forall x\ P(x)) \Rightarrow (\forall y\ Q(y)) \equiv \sim(\forall x\ P(x))\ v\ (\forall y\ Q(y))$

      $\equiv \exists x\ \sim P(x)\ v\ \forall y\ Q(y)$
    - Eliminate true existential quantifier by Skolemization
    - For true universally quantified variables, treat them as such without quantifiers
  - How to convert to CNF (similar to PL but need to work with quantifiers)

# Conversion procedure

***step* 1:** remove all “=>” and “<=>” operators

(using P => Q≡ ~P v Q and P <=> Q≡ P => Q ^ Q => P)

***step* 2:** move all negation signs to individual predicates

(using de Morgan’s law)

***step* 3:** remove all existential quantifiers ∃y

case 1: y is not in the scope of any universally quantified variable,
then replace all occurrences of y by a skolem constant

case 2: if y is in scope of universally quantified variables x1, ... xi,
then replace all occurrences of y by a skolem function

***step* 4:** remove all universal quantifiers ∀x (with the understanding that all remaining variables are universally quantified)

***step* 5:** convert the sentence into CNF (using distribution law, etc)

***step* 6:** use parenthesis to separate all disjunctions, then drop all v’s and ^’s

# Conversion examples

**$\forall$x (P(x) ^ Q(x) => R(x))**

$\forall$x ~(P(x) ^ Q(x)) v R(x) (by step 1)

$\forall$x ~P(x) v ~Q(x) v R(x) (by step 2)

~P(x) v ~Q(x) v R(x) (by step 4)

**(~P(x), ~Q(x), R(x))** (by step 6)

**$\exists$y rose(y) ^ yellow(y)**

rose(c) ^ yellow(c)

(where c is a skolem constant)

**(rose(c)), (yellow(c))**

**$\forall$x [person(x) => $\exists$y (person(y) ^ father(y, x))]**

$\forall$x [~person(x) v $\exists$y (person(y) ^ father(y, x))] (by step 1)

$\forall$x [~person(x) v (person(f_sk(x)) ^ father(f_sk(x), x))] (by step 3)

~person(x) v (person(f_sk(x)) ^ father(f_sk(x), x)) (by step 4)

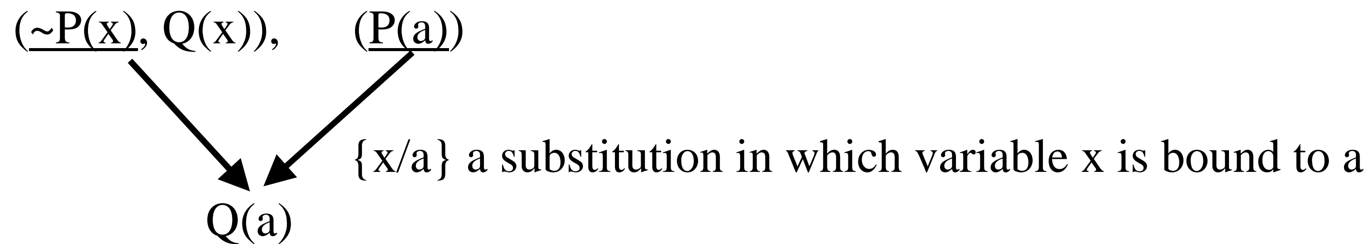
(~person(x) v person(f_sk(x)) ^ (~person(x) v father(f_sk(x), x)) (by step 5)

**(~person(x), person(f_sk(x)), (~person(x), father(f_sk(x), x))** (by step 6)

(where f_sk(.) is a skolem function)

# Unification of two clauses

- Basic idea: ∀**x P(x) => Q(x), P(a) |-- Q(a)**




  - The goal is to find a set of variable bindings so that the argument lists of two opposite literals (in two clauses) can be made the same.
  - Only variables can be bound to other things.
    - a and b cannot be unified (different constants in general refer to different objects)
    - a and f(x) cannot be unified (unless the inverse function of f is known, which is not the case for general functions in FOL)
    - f(x) and g(y) cannot be unified (function symbols f and g in general refer to different functions and their exact definitions are different in different interpretations)

- Cannot bind variable x to y if x appears anywhere in y
  - Try to unify x and f(x). If we bind x to f(x) and apply the binding to both x and f(x), we get f(x) and f(f(x)) which are still not the same (and will never be made the same no matter how many times the binding is applied)
- Otherwise, bind variable x to y, written as x/y (this guarantees to find the most general unifier, or **mgu**)
  - Suppose both x and y are variables, then they can be made the same by binding both of them to any constant c or any function f(.). Such bindings are less general and impose unnecessary restriction on x and y.
- To unify two terms of the same function symbol, unify their argument lists (**unification is recursive**)

  Ex: to unify f(x) and f(g(b)), we need to unify x and g(b)

- When the argument lists contain multiple terms, unify each pair of terms

  Ex. To unify (x, f(x), ...) (a, y, ...)

  1. unify x and a ($\theta = \{x/a\}$)
  2. apply $\theta$ to the remaining terms in both lists, resulting (f(a), ...) and (y, ...)
     1. unify f(a) and y with binding y/f(a)
     2. apply the new binding y/f(a) to $\theta$
     3. add y/f(a) to new $\theta$

# Unification Examples

- parents(x, father(x), mother(Bill)) and parents(Bill, father(Bill), y)
  - unify x and Bill: $\theta = \{x/Bill\}$
  - unify father(Bill) and father(Bill): $\theta = \{x/Bill\}$
  - unify mother(Bill) and y: $\theta = \{x/Bill\}, /mother(Bill)\}$
- parents(x, father(x), mother(Bill)) and parents(Bill, father(y), z)
  - unify x and Bill: $\theta = \{x/Bill\}$
  - unify father(Bill) and father(y): $\theta = \{x/Bill, y/Bill\}$
  - unify mother(Bill) and z: $\theta = \{x/Bill, y/Bill, z/mother(Bill)\}$
- parents(x, father(x), mother(Jane)) and parents(Bill, father(y), mother(y))
  - unify x and Bill: $\theta = \{x/Bill\}$
  - unify father(Bill) and father(y): $\theta = \{x/Bill, y/Bill\}$
  - unify mother(Jane) and mother(Bill): Failure because Jane and Bill are different constants

# More Unification Examples

- P(x, g(x), h(b)) and P(f(u, a), v, u))
  - unify x and f(u, a): $\theta$ = {x/ f(u, a)};
    remaining lists: (g(f(u, a)), h(b)) and (v, u)
  - unify g(f(u, a)) and v: $\theta$ = {x/f(u, a), v/g(f(u, a))};
    remaining lists: (h(b)) and (u)
  - unify h(b) and u: $\theta$ = {x/f(h(b), a), v/g(f(h(b), a)), u/h(b)};
- P(f(x, a), g(x, b)) and P(y, g(y, b))
  - unify f(x, a) and y: $\theta$ = {y/f(x, a)}
    remaining lists: (g(x, b)) and (g(f(x, a), b))
  - unify x and f(x, a): failure because x is in f(x, a)

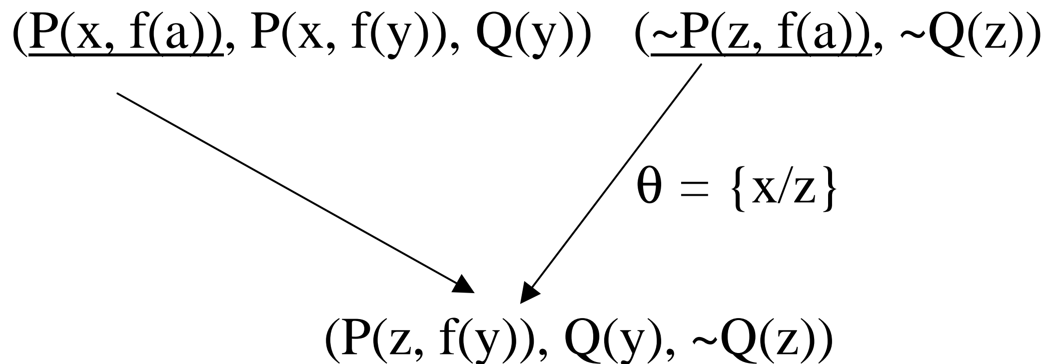
# Unification Algorithm (pp. 302-303, Chapter 10)

```
procedure unify(p, q, θ)        /* p and q are two lists of terms and |p| = |q| */
  if p = empty then return θ;  /* success */
  let r = first(p) and s = first(q);
  if r = s then return unify(rest(p), rest(q), θ);
  if r is a variable then temp = unify-var(r, s);
  else if s is a variable then temp = unify-var(s, r);
     else if both r and s are functions of the same function name then
          temp = unify(arglist(r), arglist(s), empty);
          else return "failure";
  if temp = "failure" then return "failure";  /* p and q are not unifiable */
  else θ = subst(θ, temp) ∪ temp;   /* apply tmp to old q then insert it into q */
       return unify(subst(rest(p), tmp), subst(rest(q), tmp), θ);
end{unify}
procedure unify-var(x, y)
  if x appears anywhere in y then return "failure";
  else return (x/y)
end{unify-var}
```

# Resolution in FOL

- Convert all sentences in KB (axioms, definitions, and known facts) and the goal sentence (the theorem to be proved) to clause form
- Two clauses C1 and C2 can be resolved if and only if **r** in C1 and **s** in C2 are two opposite literals, and their argument list arglist_r and arglist_s are unifiable with mgu = ***q***.
- Then derive the resolvent sentence: subst((C1 – {r}, C2 – {s}), ***q***)

  **(substitution is applied to all literals in C1 and C2, but not to any other clauses)**
- Example

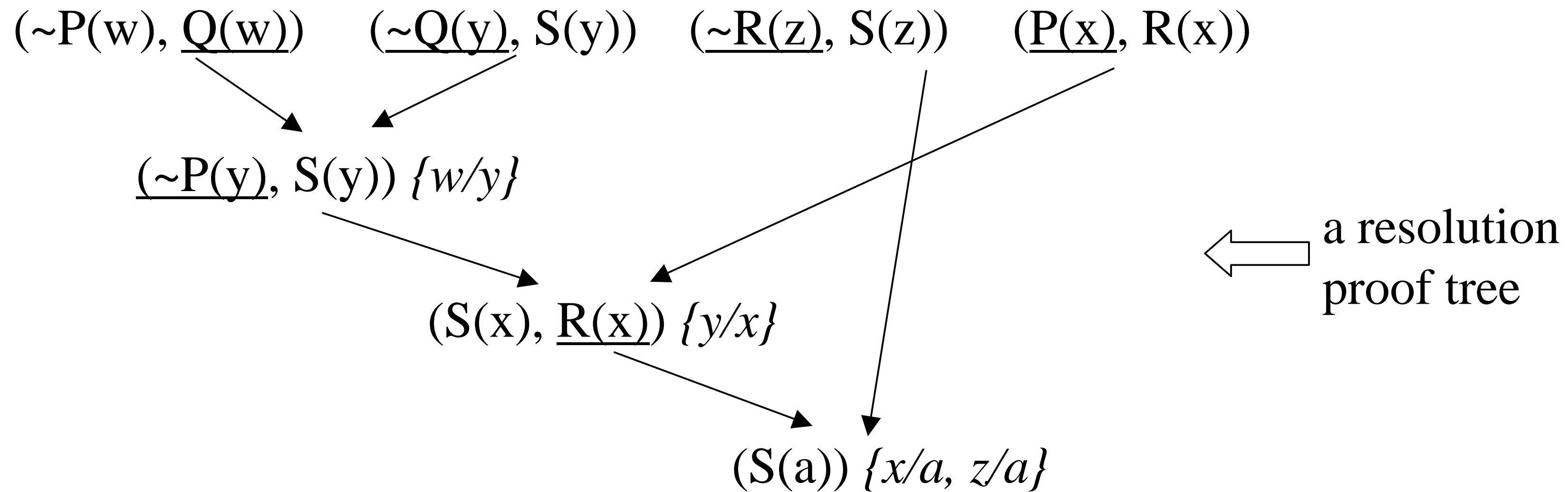


# Resolution example

- Prove that

∀w P(w) => Q(w), ∀y Q(y) => S(y), ∀z R(z) => S(z), ∀x P(x) v R(x) |= ∃u S(u)

- Convert these sentences to clauses (∃u S(u) skolemized to S(a))
- Apply resolution




- Problems
  - The theorem S(a) does not actively participate in the proof
  - Hard to determine if a proof (with consistent variable bindings) is completed if the theorem consists of more than one clause

# Resolution Refutation: a better proof strategy

- Given a consistent set of axioms KB and goal sentence Q, show that KB |= Q.
- **Proof by contradiction**: Add ~Q to KB and try to prove false.

  because (KB |= Q) <=> (KB ^ ~Q |= False, or KB ^ ~Q is inconsistent)
- How to represent “**false**” in clause form
  - P(x) ^ ~P(y) is inconsistent
  - Convert them to clause form then apply resolution

    (P(x))  (~P(y))

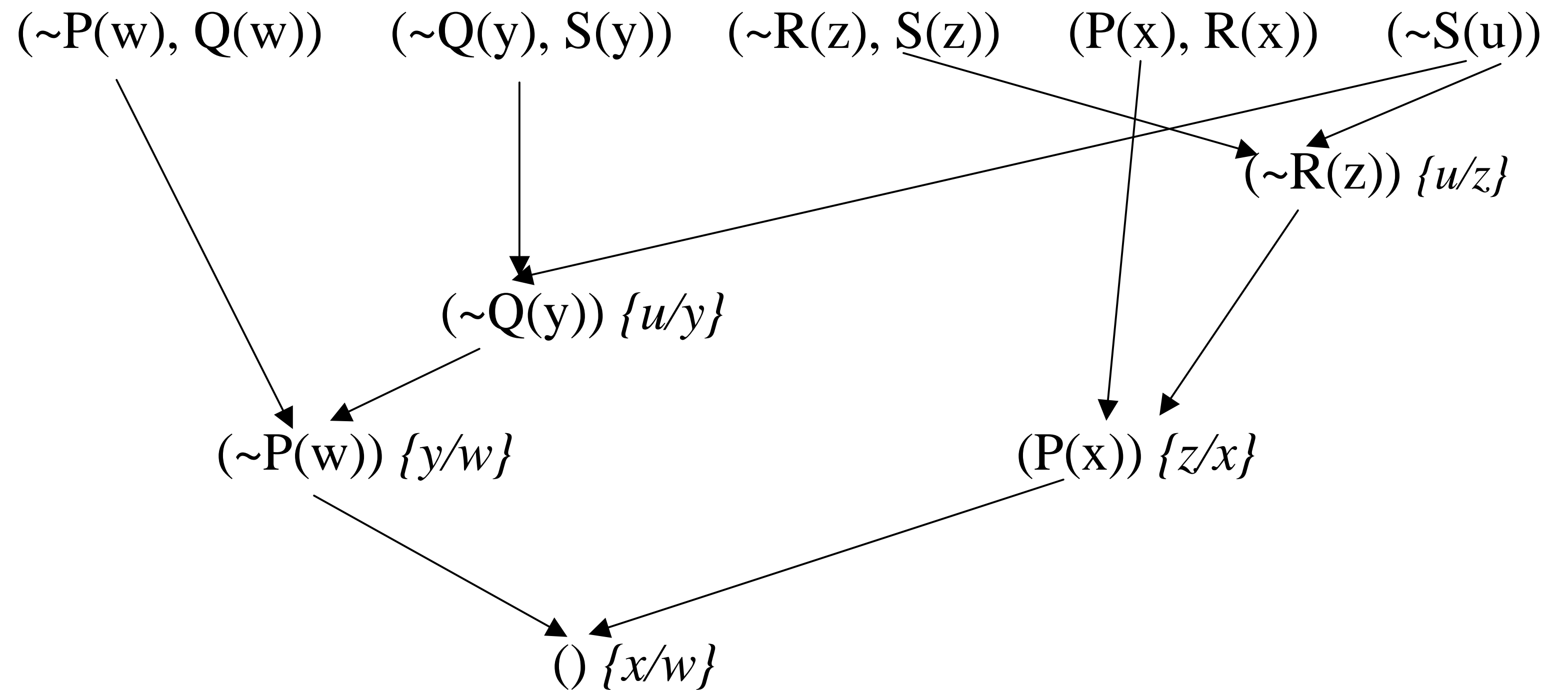
    {x/y}

    () a null clause
  - A null clause represents false (inconsistence/contradiction)
  - KB |= Q if we can derive a null clause from KB ^ ~Q by resolution

- Prove by resolution refutation that

$\forall$w P(w) => Q(w), $\forall$y Q(y) => S(y), $\forall$z R(z) => S(z), $\forall$x P(x) v R(x) |= $\exists$u S(u)

- Convert these sentences to clauses (~ $\exists$u S(u) becomes ~S(u))



# Refutation Resolution Procedure

```
procedure resolution(KB, Q)
  /* KB is a set of consistent, true FOL sentences, Q is a goal sentence.
     It  returns success if KB |-- Q, and failure otherwise */
  KB = clause(union(KB, {~Q}))   /* convert KB and ~Q to clause form */
  while null clause is not in KB do
      pick 2 sentences, S1 and S2, in KB that contain a pair of opposite
          literals whose argument lists are unifiable
      if none can be found then return "failure"
      resolvent = resolution-rule(S1, S2)
      KB = union(KB, {resolvent})
  return "success "
end{resolution}
```

# Control Strategies

- At any given time, there are multiple pairs of clauses that are resolvable. Therefore, we need a systematic way to select one such pair at each step of proof
  - May lead to a null clause
  - Without losing potentially good threads (of inference)
- There are a number of general (domain independent) strategies that are useful in controlling a resolution theorem prover.
- We'll briefly look at the following
  - Breadth first
  - Set of support
  - Unit resolution
  - Input Resolution
  - Ordered resolution
  - Subsumption

# Breadth first

- Level 0 clauses are those from the original KB and the negation of the goal.
- Level k clauses are the resolvents computed from two clauses, one of which must be from level k-1 and the other from any earlier level.
- Compute all level 1 clauses possible, then all possible level 2 clauses, etc.
- Complete, but very inefficient.

# Set of Support

- At least one parent clause must be from the negation of the goal or one of the "descendents" of such a goal clause (i.e., derived from a goal clause).
- Complete (assuming all possible set-of-support clauses are derived)
- Gives a goal directed character to the search

## Unit Resolution

- At least one parent clause must be a "unit clause," i.e., a clause containing a single literal.
- Not complete in general, but complete for Horn clause KBs

## Input Resolution

- At least one parent from the set of original clauses (from the axioms and the negation of the goal)
- Not complete in general, but complete for Horn clause KBs

## Linear Resolution

- Is an extension of Input Resolution
- use P and Q if P is in its initial KB (and query) or P is an ancestor of Q.
- Complete.

## Ordered Resolution

- Do them in order (Left to right)
- This is how Prolog operates
- Do the first element in the sentence first.
- This forces the user to define what is important in generating the "code."
- The way the sentences are written controls the resolution.

## Subsumption

- Eliminate all clauses that are subsumed (more specific than) by an existing clause to keep the KB small.
- Like factoring, this is just removing things that merely clutter up the space and will not affect the final result.
- I.e. if P(x) is already in the KB, adding P(A) makes no sense -- P(x) is a superset of P(A).
- Likewise adding P(A) v Q(B) would add nothing to the KB either.

# Example of Automatic Theorem Proof:

***Did Curiosity kill the cat***

- Jack owns a dog. Every dog owner is an animal lover. No animal lover kills an animal. Either Jack or Curiosity killed the cat, who is named Tuna. Did Curiosity kill the cat?
- These can be represented as follows:

  A. (∃x) Dog(x) ^ Owns(Jack,x)

  B. (∀x) ((∃y) Dog(y) ^ Owns(x, y)) => AnimalLover(x)

  C. (∀x) AnimalLover(x) => (∀y) Animal(y) => ~Kills(x,y)

  D. Kills(Jack,Tuna) v Kills(Curiosity,Tuna)

  E. Cat(Tuna)

  F. (∀x) Cat(x) => Animal(x)

  Q. Kills(Curiosity, Tuna)

- **Convert to clause form**

  A1. (Dog(D)) /* D is a skolem constant */

  A2. (Owns(Jack,D))

  B. (~Dog(y), ~Owns(x, y), AnimalLover(x))

  C. (~AnimalLover(x), ~Animal(y), ~Kills(x,y))

  D. (Kills(Jack,Tuna), Kills(Curiosity,Tuna))

  E. Cat(Tuna)

  F. (~Cat(x), Animal(x))

- **Add the negation of query:**

  Q: (~Kills(Curiosity, Tuna))

- **The resolution refutation proof**

  R1: Q, D, {}, (Kills(Jack, Tuna))

  R2: R1, C, {x/Jack, y/Tuna}, (~AnimalLover(Jack), ~Animal(Tuna))

  R3: R2, B, {x/Jack}, (~Dog(y), ~Owns(Jack, y), ~Animal(Tuna))

  R4: R3, A1, {y/D}, (~Owns(Jack, D), ~Animal(Tuna))

  R5: R4, A2, {}, (~Animal(Tuna))

  R6: R5, F, {x/Tuna}, (~Cat(Tuna))

  R7: R6, E, {} ()

# Horn Clauses

- A Horn clause is a clause with at most one positive literal:

  (~P1(x), ~P2(x), ..., ~Pn(x) v Q(x)),   equivalent to

  ∀x P1(x) ^ P2(x) ... ^ Pn(x) => Q(x)   or

  Q(x) <= P1(x), P2(x), ... , Pn(x)   (in prolog format)

  – if contains no negated literals (i.e., Q(a) <=): facts
  – if contains no positive literals (<= P1(x), P2(x), ... , Pn(x)): query
  – if contain no literal at all (<=): null clause
- Most knowledge can be represented by Horn clauses
- Easier to understand (keeps the implication form)
- Easier to process than FOL
- Horn clauses represent a subset of the set of sentences representable in FOL (e.g., it cannot represent uncertain conclusions, e.g.,

  Q(x) v R(x) <= P(x)).

# Logic Programming

- Resolution with Horn clause is like a function all:

Q(x) <= P1(x), P2(x), ... , Pn(x)

Function name

Function body

Q(x) <= P1(x), P2(x), ... , Pn(x)        <= Q(a)

θ

<= P1(a), P2(a), ... , Pn(a)

Unification is like parameter passing

To solve Q(a), we solve P1(a), P2(a), ... , and Pn(a). This is called problem reduction (P1(a), ... Pn(a) are subgoals).

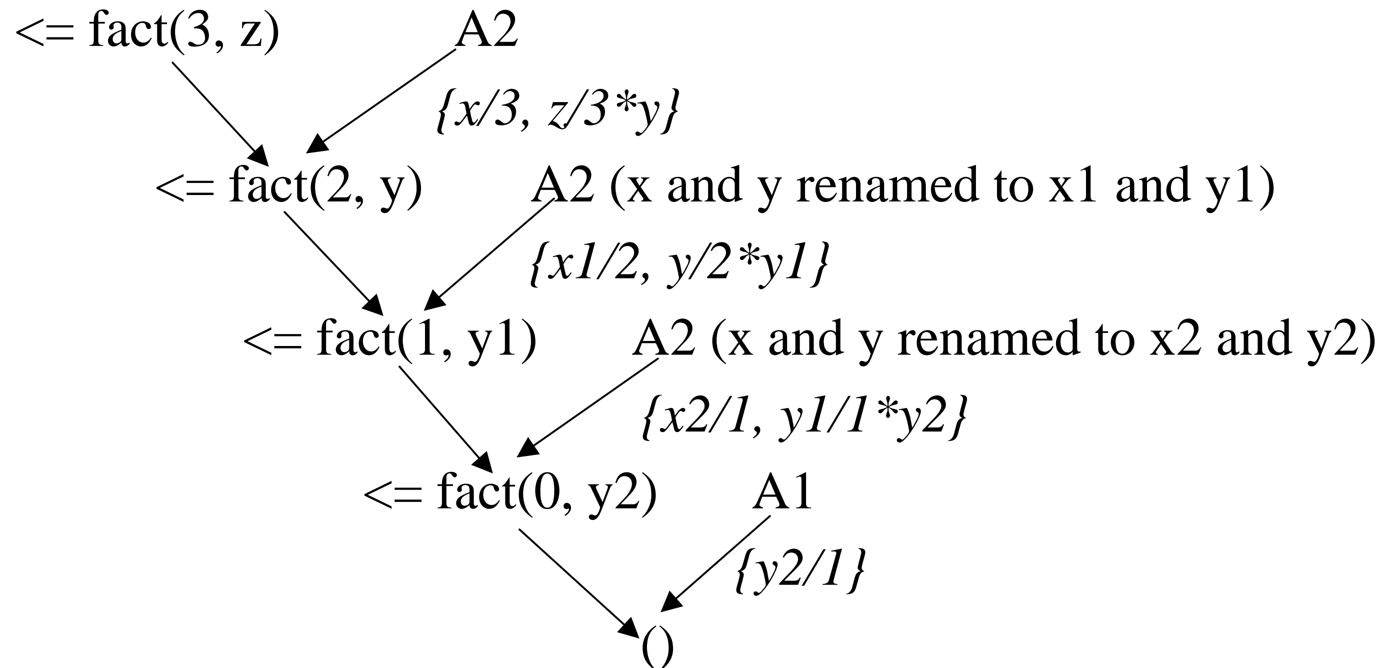
We then continue to call functions to solve P1(a), ..., by resolving <= P1(a), P2(a), ... , Pn(a) with clauses P(y) <= R1(y), ... Rm(y), etc.

# Example of Logic Programming

***Computing factorials***

A1: fact(0, 1) <= /* base case: 0! = 1 */

A2: fact(x, x*y) <= fact(x-1, y) /* recursion: x! = x*(x-1)! */




Extract answer from the variable bindings:

z = 3*y = 3*2*y1 = 3*2*1*y2 = 3*2*1*1 = 6

# Prolog

- A logic programming language based on Horn clauses
  - Resolution refutation
  - Control strategy: goal directed and depth-first
    - always start from the goal clause,
    - always use the new resolvant as one of the parent clauses for resolution
    - backtracking when the current thread fails
    - complete for Horn clause KB
  - Support answer extraction (can request single or all answers)
  - Orders the clauses and literals with a clause to resolve non-determinism
    - Q(a) may match both Q(x) <= P(x) and Q(y) <= R(y)
    - A (sub)goal clause may contain more than one literals, i.e., <= P1(a), P2(a)
  - Use "closed world" assumption (negation as failure)
    - If it fails to derive P(a), then assume ~P(a)

# Other issues

- FOL is semi-decidable
  - We want to answer the question if KB |= S
  - If actually KB |= S (or KB |= ~S), then a complete proof procedure will terminate with a positive (or negative) answer within finite steps of inference
  - If neither S nor ~S logically follows KB, then there is **no** proof procedure will terminate within **finite** steps of inference for **arbitrary** KB and S.
  - The semi-decidability is caused by
    - infinite domain and incomplete axiom set (knowledge base)
    - Ex: KB contains only one clause fact(x, x*y) <= fact(x-1, y). To prove fact(3, z) will run forever
  - By Godel's Incomplete Theorem, no logical system can be complete (e.g., no matter how many pieces of knowledge you include in KB, there is always a legal sentence S such that neither S nor ~S logically follow KB).
  - Closed world assumption is a practical way to circumvent this problem, but it make the logical system non-monotonic, therefore non-FOL

- Forward chaning
  - Proof starts with the new fact **P(a) <=,** (often case specific data)
  - Resolve it with rules **Q(x) <= P(x)** to derived new fact **Q(a) <=**
  - Additional inference is then triggered by **Q(a)** <=, etc. The process stops when the theorem intended to proof (if there is one) has been generated or no new sentenced can be generated.
  - Implication rules are always used in the way of modus ponens (*from premises to conclusions*), i.e., in the direction of implication arrows
  - This defines a forward chaining inference procedure because it moves "forward" from fact oward the goal (also called data driven).

- Backward chanining
  - Proof starts with the goal query (theorem to be proven) **<= Q(a)**
  - Resolve it with rules **Q(x) <= P(x)** to derived new query **<= P(a)**
  - Additional inference is then triggered by **<= P(a)**, etc. The process stops when a null clause is derived.
  - Implication rules are always used in the way of modus tollens (*from conclusions to premises*), i.e., in the reverse direction of implication arrows
  - This defines a backward chaining inference procedure because it moves “backward" from the goal (also called goal driven).
  - Backward chaining is more efficient than forward chaining as it is more focused. However, it requires that the goal (theorem to be proven) be known prior to the inference