Logical Inference 3 resolution

Chapter 9

Some material adopted from notes by Andreas Geyer-Schulz,, Chuck Dyer, and Mary Getoor

Resolution

- Resolution is a **sound** and **complete** inference procedure for unrestricted FOL
- Reminder: Resolution rule for propositional logic:
  - $P_1 \vee P_2 \vee \ldots \vee P_n$
  - $\neg P_1 \vee Q_2 \vee \ldots \vee Q_m$
  - Resolvent: $P_2 \vee \ldots \vee P_n \vee Q_2 \vee \ldots \vee Q_m$
- We'll need to extend this to handle quantifiers and variables

Two Common Normal Forms for a KB

Implicative normal form

- Set of sentences where each is expressed as an implication
- Left hand side of implication is a conjunction of 0 or more literals
- P , Q, P ∧ Q => R

Conjunctive normal form

- Set of sentences where each is a disjunction of atomic literals
- P, Q, ~P ∨ ~Q ∨ R

Resolution covers many cases

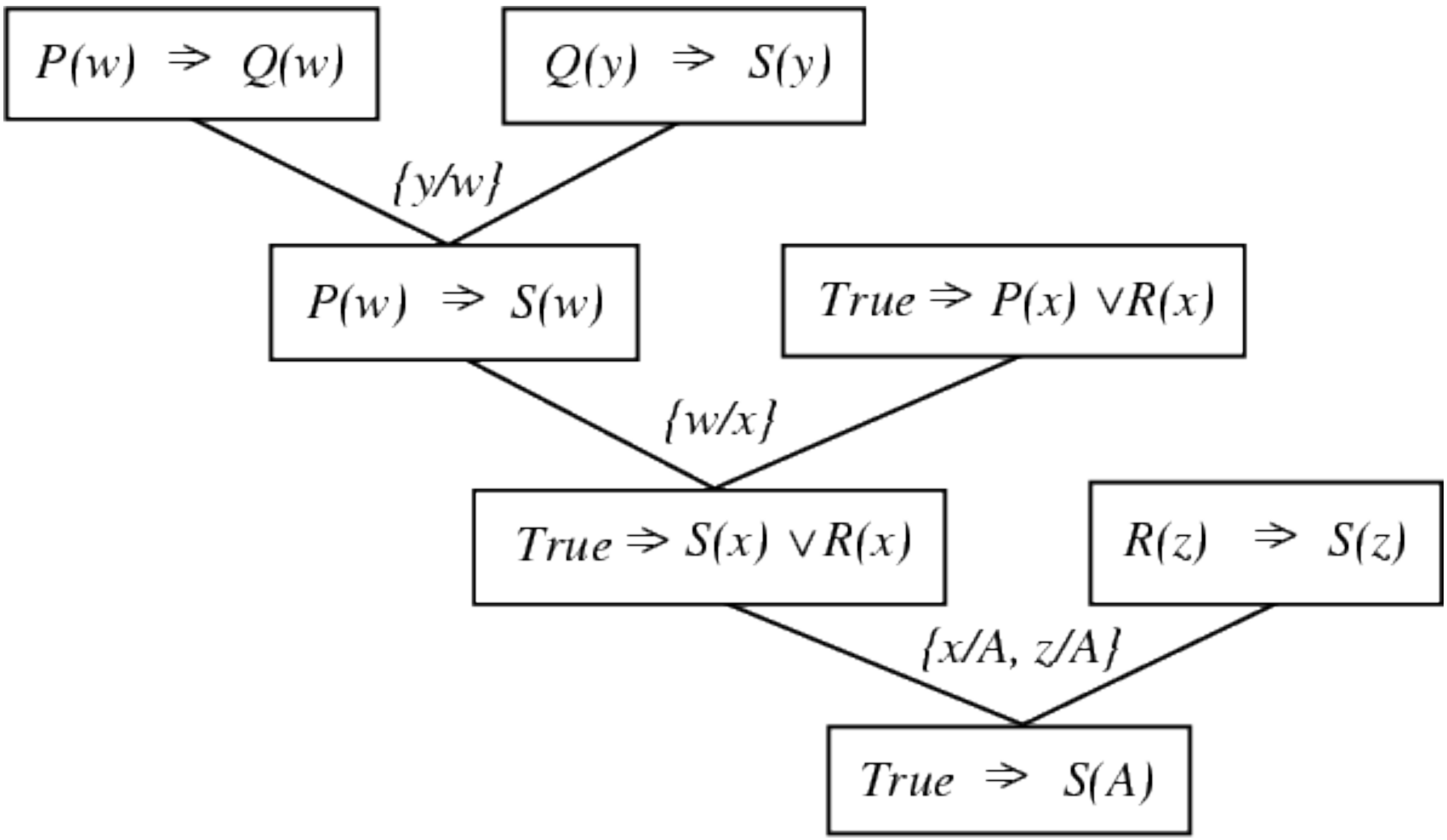
- Modes Ponens
  - from P and P $\rightarrow$ Q derive Q
  - from P and $\neg$ P $\vee$ Q derive Q
- Chaining
  - from P $\rightarrow$ Q and Q $\rightarrow$ R derive P $\rightarrow$ R
  - from ($\neg$ P $\vee$ Q) and ($\neg$ Q $\vee$ R) derive $\neg$ P $\vee$ R
- Contradiction detection
  - from P and $\neg$ P derive false
  - from P and $\neg$ P derive the empty clause (=false)

Resolution in first-order logic

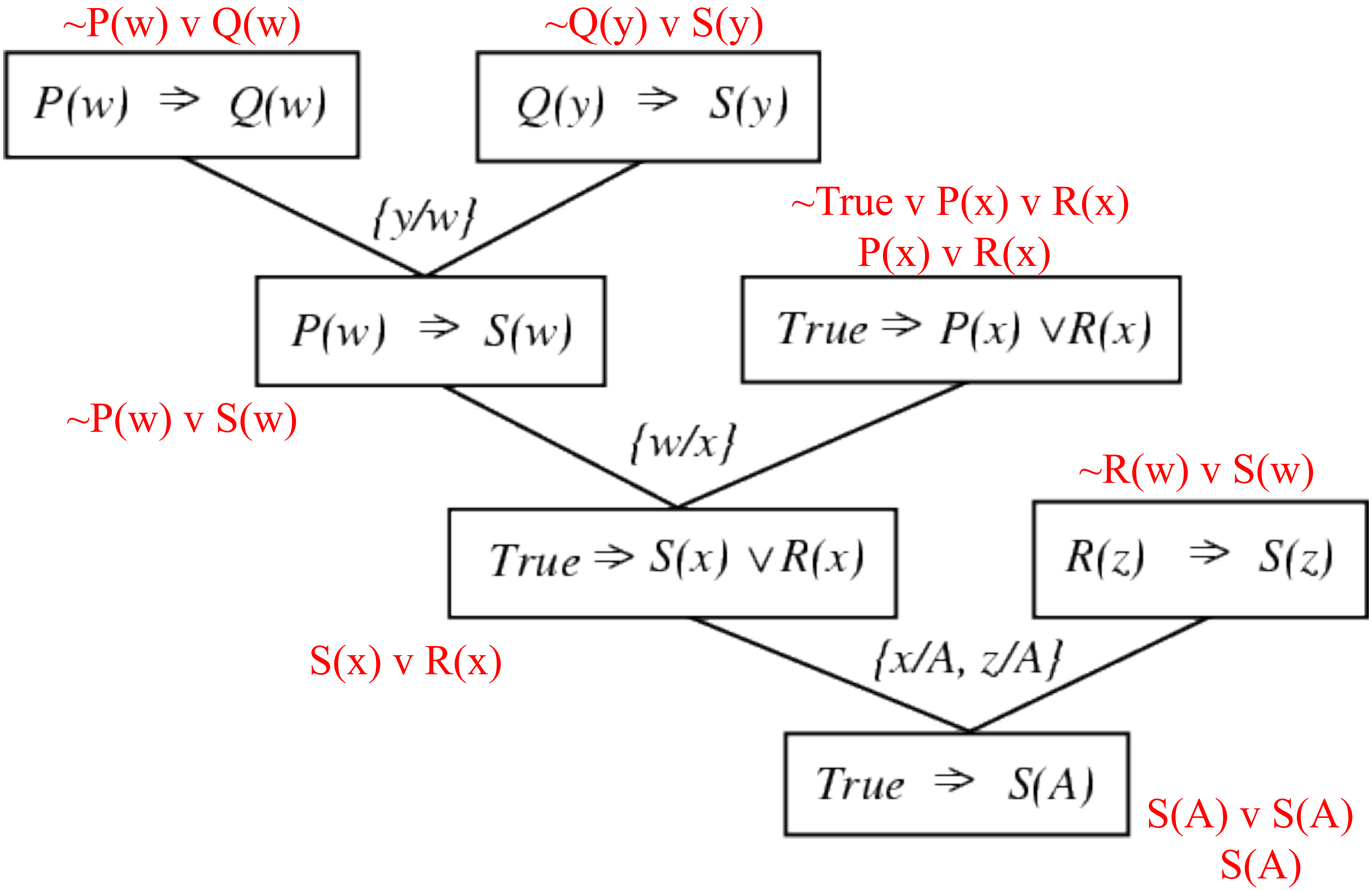
- Given sentences in *conjunctive normal form:*
  - $P_1 \vee \ldots \vee P_n$ and $Q_1 \vee \ldots \vee Q_m$
  - P_i and Q_i are literals, i.e., positive or negated predicate symbol with its terms
- if P_j and $\neg Q_k$ **unify** with substitution list θ, then derive the resolvent sentence:

  $\text{subst}(\theta, P_1 \vee \ldots \vee P_{j-1} \vee P_{j+1} \ldots P_n \vee Q_1 \vee \ldots Q_{k-1} \vee Q_{k+1} \vee \ldots \vee Q_m)$
- Example
  - from clause $\mathbf{P(x, f(a)) \vee P(x, f(y)) \vee Q(y)}$
  - and clause $\mathbf{\neg P(z, f(a)) \vee \neg Q(z)}$
  - derive resolvent $\mathbf{P(z, f(y)) \vee Q(y) \vee \neg Q(z)}$
  - Using $\boldsymbol{\theta} = \mathbf{\{x/z\}}$

A resolution proof tree



A resolution proof tree



Resolution refutation (1)

- 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, i.e.:

  (KB |- Q) ↔ (KB ∧ ¬Q |- False)

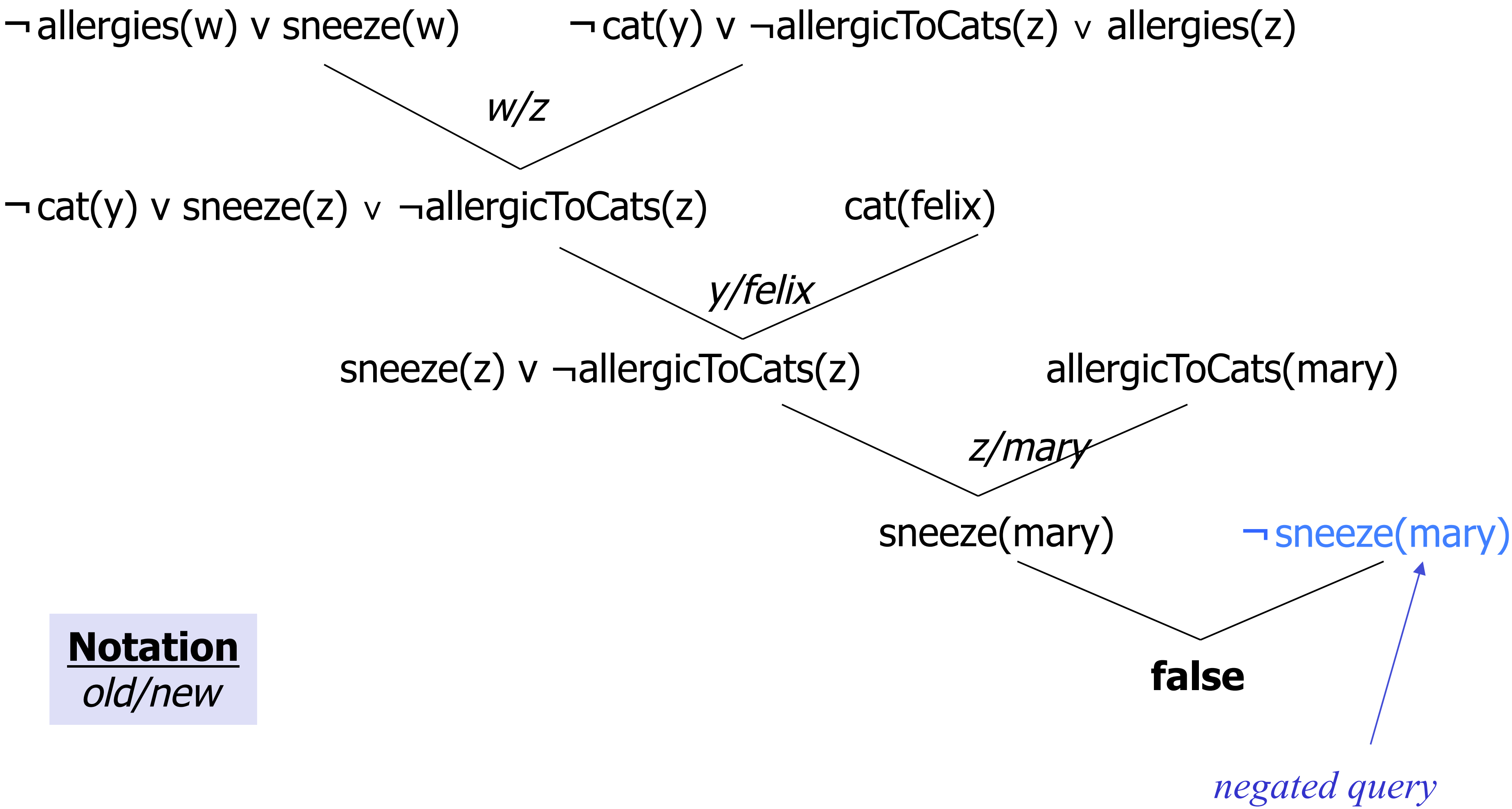
Resolution refutation (2)

- Resolution is **refutation complete**: it can establish that a given sentence Q is entailed by KB, but can't always generate all consequences of a set of sentences
- It cannot be used to prove that Q is **not entailed** by KB
- Resolution **won't always give an answer** since entailment is only semi-decidable
  - And you can't just run two proofs in parallel, one trying to prove Q and the other trying to prove ¬Q, since KB might not entail either one

Resolution example

- KB:
  - allergies(X) → sneeze(X)
  - cat(Y) ∧ allergicToCats(X) → allergies(X)
  - cat(felix)
  - allergicToCats(mary)
- Goal:
  - sneeze(mary)

Refutation resolution proof tree



Notation

old/new

Questions to be answered

- How to convert FOL sentences to conjunctive normal form (a.k.a. CNF, clause form): **normalization and skolemization**
- How to unify two argument lists, i.e., how to find their most general unifier (**mgu**) q: **unification**
- How to determine which two clauses in KB should be resolved next (among all resolvable pairs of clauses) : **resolution (search) strategy**

Converting to CNF

Converting sentences to CNF

1. Eliminate all ↔ connectives

   $(P \leftrightarrow Q) \Rightarrow ((P \rightarrow Q) \wedge (Q \rightarrow P))$

2. Eliminate all → connectives

   $(P \rightarrow Q) \Rightarrow (\neg P \vee Q)$

3. Reduce the scope of each negation symbol to a single predicate

   $\neg\neg P \Rightarrow P$

   $\neg(P \vee Q) \Rightarrow \neg P \wedge \neg Q$

   $\neg(P \wedge Q) \Rightarrow \neg P \vee \neg Q$

   $\neg(\forall x)P \Rightarrow (\exists x)\neg P$

   $\neg(\exists x)P \Rightarrow (\forall x)\neg P$

4. Standardize variables: rename all variables so that each quantifier has its own unique variable name

See the function to_cnf() in logic.py

Converting sentences to clausal form

Skolem constants and functions

5. Eliminate existential quantification by introducing Skolem constants/functions

$(\exists x)P(x) \Rightarrow P(C)$

C is a Skolem constant (a brand-new constant symbol that is not used in any other sentence)

$(\forall x)(\exists y)P(x,y) \Rightarrow (\forall x)P(x, f(x))$

since ∃ is within scope of a universally quantified variable, use a **Skolem function f** to construct a new value that **depends on** the universally quantified variable

f must be a brand-new function name not occurring in any other sentence in the KB

E.g., $(\forall x)(\exists y)loves(x,y) \Rightarrow (\forall x)loves(x,f(x))$

In this case, f(x) specifies the person that x loves

a better name might be **oneWhoIsLovedBy**(x)

Converting sentences to clausal form

6. Remove universal quantifiers by (1) moving them all to the left end; (2) making the scope of each the entire sentence; and (3) dropping the "prefix" part

   Ex: $(\forall x)P(x) \Rightarrow P(x)$

7. Put into conjunctive normal form (conjunction of disjunctions) using distributive and associative laws

   $(P \wedge Q) \vee R \Rightarrow (P \vee R) \wedge (Q \vee R)$

   $(P \vee Q) \vee R \Rightarrow (P \vee Q \vee R)$

8. Split conjuncts into separate clauses
9. Standardize variables so each clause contains only variable names that do not occur in any other clause

An example

$$(\forall x)(P(x) \rightarrow ((\forall y)(P(y) \rightarrow P(f(x,y))) \wedge \neg(\forall y)(Q(x,y) \rightarrow P(y))))$$

2. Eliminate $\rightarrow$

$$(\forall x)(\neg P(x) \vee ((\forall y)(\neg P(y) \vee P(f(x,y))) \wedge \neg(\forall y)(\neg Q(x,y) \vee P(y))))$$

3. Reduce scope of negation

$$(\forall x)(\neg P(x) \vee ((\forall y)(\neg P(y) \vee P(f(x,y))) \wedge (\exists y)(Q(x,y) \wedge \neg P(y))))$$

4. Standardize variables

$$(\forall x)(\neg P(x) \vee ((\forall y)(\neg P(y) \vee P(f(x,y))) \wedge (\exists z)(Q(x,z) \wedge \neg P(z))))$$

5. Eliminate existential quantification

$$(\forall x)(\neg P(x) \vee ((\forall y)(\neg P(y) \vee P(f(x,y))) \wedge (Q(x,g(x)) \wedge \neg P(g(x)))))$$

6. Drop universal quantification symbols

$$(\neg P(x) \vee ((\neg P(y) \vee P(f(x,y))) \wedge (Q(x,g(x)) \wedge \neg P(g(x)))))$$

Example

7. Convert to conjunction of disjunctions

   $(\neg P(x) \vee \neg P(y) \vee P(f(x,y))) \wedge (\neg P(x) \vee Q(x,g(x))) \wedge$

   $(\neg P(x) \vee \neg P(g(x)))$

8. Create separate clauses

   $\neg P(x) \vee \neg P(y) \vee P(f(x,y))$

   $\neg P(x) \vee Q(x,g(x))$

   $\neg P(x) \vee \neg P(g(x))$

9. Standardize variables

   $\neg P(x) \vee \neg P(y) \vee P(f(x,y))$

   $\neg P(z) \vee Q(z,g(z))$

   $\neg P(w) \vee \neg P(g(w))$

Unification

Unification

- Unification is a **"pattern-matching"** procedure
  - Takes two atomic sentences (i.e., literals) as input
  - Returns "failure" if they do not match and a substitution list, θ, if they do
- That is, $unify(p,q) = \theta$ means $subst(\theta, p) = subst(\theta, q)$ for two atomic sentences, p and q
- **θ** is called the **most general unifier** (mgu)
- All variables in the given two literals are implicitly universally quantified
- To make literals match, replace (universally quantified) variables by terms

Unification algorithm

procedure unify(p, q, θ)
    Scan p and q left-to-right and find the first corresponding
      terms where p and q "disagree" (i.e., p and q not equal)
    If there is no disagreement, return θ (success!)
    Let r and s be the terms in p and q, respectively,
      where disagreement first occurs
    If variable(r) then {
      Let θ = union(θ, {r/s})
      Return unify(subst(θ, p), subst(θ, q), θ)
    } else if variable(s) then {
      Let θ = union(θ, {s/r})
      Return unify(subst(θ, p), subst(θ, q), θ)
    } else return "Failure"
  end

See the function unify() in logic.py

Unification: Remarks

- *Unify* is a linear-time algorithm that returns the most general unifier (mgu), i.e., the shortest-length substitution list that makes the two literals match
- In general, there isn't a **unique** minimum-length substitution list, but unify returns one of minimum length
- Common constraint: A variable can never be replaced by a term containing that variable

  Example: x/f(x) is illegal.

  - This "occurs check" should be done in the above pseudo-code before making the recursive calls

Unification examples

- Example:
  - parents(x, father(x), mother(Bill))
  - parents(Bill, father(Bill), y)
  - {x/Bill,y/mother(Bill)} yields parents(Bill,father(Bill), mother(Bill))
- Example:
  - parents(x, father(x), mother(Bill))
  - parents(Bill, father(y), z)
  - {x/Bill,y/Bill,z/mother(Bill)} yields parents(Bill,father(Bill), mother(Bill))
- Example:
  - parents(x, father(x), mother(Jane))
  - parents(Bill, father(y), mother(y))
  - Failure

Resolution example

Practice example

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?

Practice example

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. $(\exists x)\ Dog(x) \land Owns(Jack,x)$

B. $(\forall x)\ ((\exists y)\ Dog(y) \land Owns(x, y)) \rightarrow AnimalLover(x)$

C. $(\forall x)\ AnimalLover(x) \rightarrow ((\forall y)\ Animal(y) \rightarrow \neg Kills(x,y))$

D. $Kills(Jack,Tuna) \lor Kills(Curiosity,Tuna)$

E. $Cat(Tuna)$

F. $(\forall x)\ Cat(x) \rightarrow Animal(x)$

G. $Kills(Curiosity, Tuna)$ ← GOAL

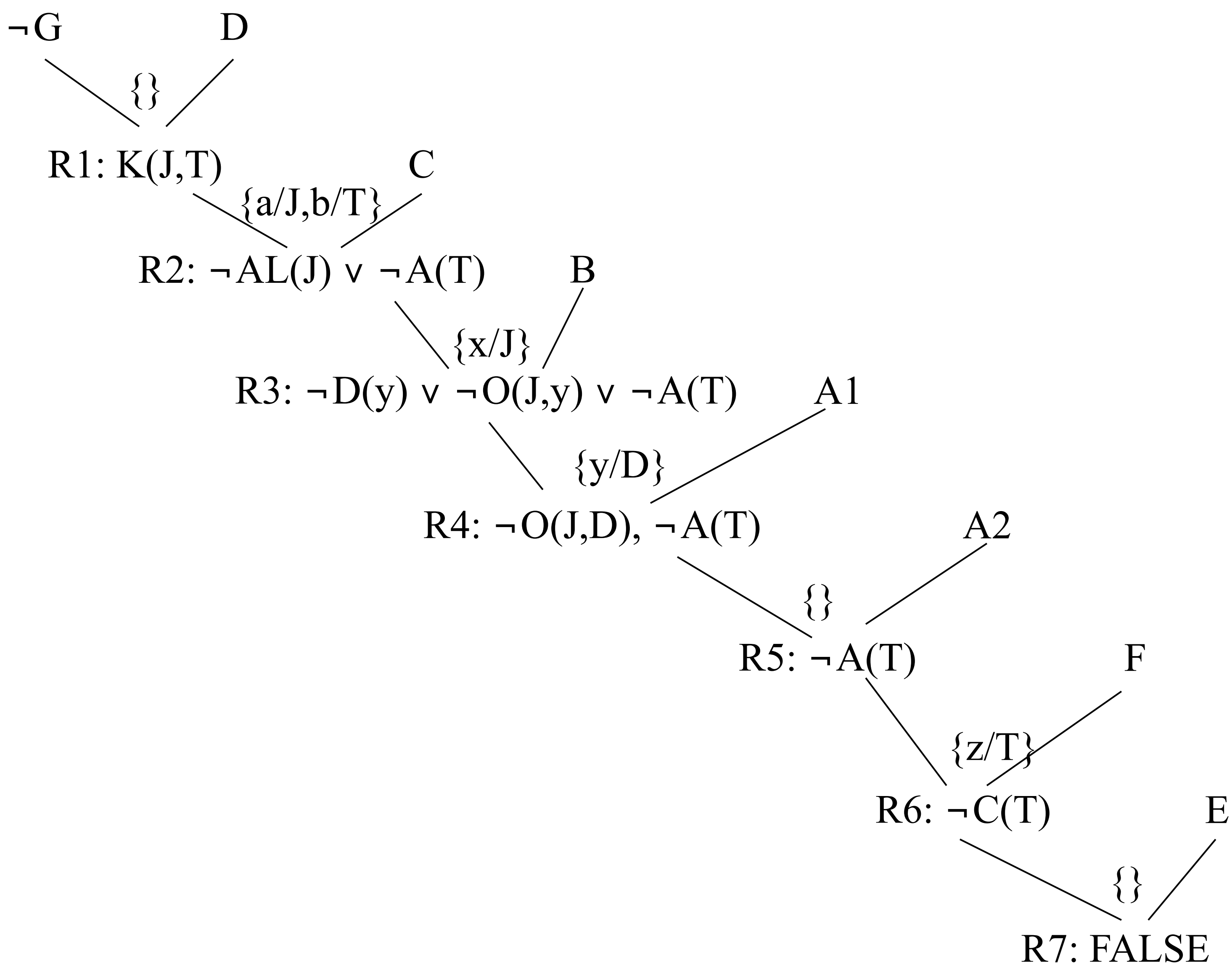
∃x Dog(x) ∧ Owns(Jack,x)
∀x (∃y) Dog(y) ∧ Owns(x, y) → AnimalLover(x)
∀x AnimalLover(x) → (∀y Animal(y) → ¬Kills(x,y))
Kills(Jack,Tuna) ∨ Kills(Curiosity,Tuna)
Cat(Tuna)
∀x Cat(x) → Animal(x)
Kills(Curiosity, Tuna)

- **Convert to clause form**
  - A1. (Dog(D))
  - A2. (Owns(Jack,D))
  - B. (¬Dog(y), ¬Owns(x, y), AnimalLover(x))
  - C. (¬AnimalLover(a), ¬Animal(b), ¬Kills(a,b))
  - D. (Kills(Jack,Tuna), Kills(Curiosity,Tuna))
  - E. Cat(Tuna)
  - F. (¬Cat(z), Animal(z))
- **Add the negation of query:**
  - ¬G: ¬Kills(Curiosity, Tuna)

The resolution refutation proof

R1: ¬G, D, {}	(Kills(Jack, Tuna))
R2: R1, C, {a/Jack, b/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, {z/Tuna}	(~Cat(Tuna))
R7: R6, E, {}	FALSE

The proof tree



Resolution search strategies

Resolution TP as search

- Resolution can be thought of as the **bottom-up construction of a search tree**, where the leaves are the clauses produced by KB and the negation of the goal
- When a pair of clauses generates a new resolvent clause, add a new node to the tree with arcs directed from the resolvent to the two parent clauses
- **Resolution succeeds** when a node containing the **False** clause is produced, becoming the **root node** of the tree
- A strategy is **complete** if its use guarantees that the empty clause (i.e., false) can be derived whenever it is entailed

Strategies

- There are a number of general (domain-independent) strategies that are useful in controlling a resolution theorem prover
- Well briefly look at the following:
  - Breadth-first
  - Length heuristics
  - Set of support
  - Input resolution
  - Subsumption
  - Ordered resolution

Example

1. Battery-OK ∧ Bulbs-OK → Headlights-Work
2. Battery-OK ∧ Starter-OK → Empty-Gas-Tank ∨ Engine-Starts
3. Engine-Starts → Flat-Tire ∨ Car-OK
4. Headlights-Work
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. Goal: Flat-Tire ?

Example

1. ¬Battery-OK v ¬Bulbs-OK v Headlights-Work
2. ¬Battery-OK v ¬Starter-OK v Empty-Gas-Tank v Engine-Starts
3. ¬Engine-Starts v Flat-Tire v Car-OK
4. Headlights-Work
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. ¬Flat-Tire ← **negated goal**

Breadth-first search

- Level 0 clauses are the original axioms 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 possible level 1 clauses, then all possible level 2 clauses, etc.
- Complete, but very inefficient

BFS example

1. ¬Battery-OK ∨ ¬Bulbs-OK ∨ Headlights-Work
2. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
3. ¬Engine-Starts ∨ Flat-Tire ∨ Car-OK
4. Headlights-Work
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. ¬Flat-Tire

1,4 10. ¬Battery-OK ∨ ¬Bulbs-OK

1,5 11. ¬Bulbs-OK ∨ Headlights-Work

2,3 12. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Flat-Tire ∨ Car-OK

2,5 13. ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts

2,6 14. ¬Battery-OK ∨ Empty-Gas-Tank ∨ Engine-Starts

2,7 15. ¬Battery-OK ¬ Starter-OK ∨ Engine-Starts

16. … [and we’re still only at Level 1!]

Length heuristics

- **Shortest-clause heuristic**:
  Generate a clause with the fewest literals first
- **Unit resolution**:
  Prefer resolution steps in which at least one parent clause is a "unit clause," i.e., a clause containing a single literal
  - Not complete in general, but complete for Horn clause KBs

Unit resolution example

1. ¬Battery-OK ∨ ¬Bulbs-OK ∨ Headlights-Work
2. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
3. ¬Engine-Starts ∨ Flat-Tire ∨ Car-OK
4. Headlights-Work
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. ¬Flat-Tire

1,5 10. ¬Bulbs-OK ∨ Headlights-Work

2,5 11. ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts

2,6 12. ¬Battery-OK ∨ Empty-Gas-Tank ∨ Engine-Starts

2,7 13. ¬Battery-OK ¬ Starter-OK ∨ Engine-Starts

3,8 14. ¬Engine-Starts ∨ Flat-Tire

3,9 15. ¬Engine-Starts ¬ Car-OK

16. … [this doesn’t seem to be headed anywhere either!]

Set of support

- At least one parent clause must be the negation of the goal *or* a "descendant" of such a goal clause (i.e., derived from a goal clause)
- *When there's a choice, take the most recent descendant*
- Complete, assuming all possible set-of-support clauses are derived
- Gives a goal-directed character to the search (e.g., like backward chaining)

Set of support example

1. ¬Battery-OK ∨ ¬Bulbs-OK ∨ Headlights-Work
2. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
3. ¬Engine-Starts ∨ Flat-Tire ∨ Car-OK
4. Headlights-Work
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. ¬Flat-Tire

9,3 10. ¬Engine-Starts ∨ Car-OK
10,2 11. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Car-OK
10,8 12. ¬Engine-Starts
11,5 13. ¬Starter-OK ∨ Empty-Gas-Tank ∨ Car-OK
11,6 14. ¬Battery-OK ∨ Empty-Gas-Tank ∨ Car-OK
11,7 15. ¬Battery-OK ∨ ¬Starter-OK ∨ Car-OK
16. … [a bit more focused, but we still seem to be wandering]

Unit resolution + set of support example

1. ¬Battery-OK ∨ ¬Bulbs-OK ∨ Headlights-Work
2. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
3. ¬Engine-Starts ∨ Flat-Tire ∨ Car-OK
4. Headlights-Work
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. ¬Flat-Tire

9,3 10. ¬Engine-Starts ∨ Car-OK

10,8 11. ¬Engine-Starts

11,2 12. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank

12,5 13. ¬Starter-OK ∨ Empty-Gas-Tank

13,6 14. Empty-Gas-Tank

14,7 15. FALSE

[Hooray! Now that's more like it!]

Simplification heuristics

- **Subsumption:**
  Eliminate sentences that are subsumed by (more specific than) an existing sentence to keep KB small
  - 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) ∨ Q(B) would add nothing to the KB
- **Tautology:**
  Remove any clause containing two complementary literals (tautology)
- **Pure symbol:**
  If a symbol always appears with the same "sign," remove all the clauses that contain it

Example (Pure Symbol)

1. ~~¬Battery-OK ∨ ¬Bulbs-OK ∨ Headlights-Work~~
2. ¬Battery-OK ∨ ¬Starter-OK ∨ Empty-Gas-Tank ∨ Engine-Starts
3. ¬Engine-Starts ∨ Flat-Tire ∨ Car-OK
4. ~~Headlights-Work~~
5. Battery-OK
6. Starter-OK
7. ¬Empty-Gas-Tank
8. ¬Car-OK
9. ¬Flat-Tire

Input resolution

- At least one parent must be one of the input sentences (i.e., either a sentence in the original KB or the negation of the goal)
- Not complete in general, but complete for Horn clause KBs
- Linear resolution
  - Extension of input resolution
  - One of the parent sentences must be an input sentence *or* an ancestor of the other sentence
  - Complete

Ordered resolution

- Search for resolvable sentences in order (left to right)
- This is how Prolog operates
- Resolve 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