Inference in first-order logic

Russell and Norvig Chapter 9

Outline

- Reducing first-order inference to propositional inference
- Generalized Modus Ponens
- Forward chaining
- Backward chaining
- Resolution

FOL to PL

- First order inference can be done by converting the knowledge base to PL and using propositional inference.
  - How to convert universal quantifiers?
    - Replace variable by ground term.
  - How to convert existential quantifiers?
    - Skolemization.

Universal Instantiation (UI)

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

$$\frac{\forall v\ \alpha}{\text{Subst}(\{v/g\}, \alpha)}$$

Subst – the substitution operator

for any variable v and ground term g

E.g., $\forall x\ \text{King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$ yields:

$\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John})$

$\text{King}(\text{Father}(\text{John})) \wedge \text{Greedy}(\text{Father}(\text{John})) \Rightarrow \text{Evil}(\text{Father}(\text{John}))$

Existential Instantiation (EI)

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

$$\frac{\exists v\ \alpha}{\text{Subst}(\{v/k\}, \alpha)}$$

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

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

provided C_1 is a new constant symbol, called a Skolem constant

EI versus UI

- UI can be applied several times to *add* new sentences; the new KB is logically equivalent to the old.
- EI can be applied once to replace the existential sentence.

Reduction to propositional inference

- Suppose the KB contains just the following:
  - ∀x King(x) ∧ Greedy(x) ⇒ Evil(x)
  - King(John)
  - Greedy(John)
  - Brother(Richard,John)
- Instantiating the universal sentence in all possible ways:
  - King(John) ∧ Greedy(John) ⇒ Evil(John)
  - King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)
- The new KB is propositionalized

Reduction (cont)

- *CLAIM*: A ground sentence is entailed by the new KB iff entailed by the original KB.
- *CLAIM*: Every FOL KB can be propositionalized so as to preserve entailment
- *IDEA*: propositionalize KB and query, apply resolution, return result
- *PROBLEM*: with function symbols, there are infinitely many ground terms,
  - e.g., *Father(Father(Father(John)))*
  - In our natural numbers example: NatNum(S(0)), NatNum(S(S(0))).…

The question of entailment for FOL is semidecidable: no algorithm exists that says no to every nonentailed sentence.

Reduction (cont)

- *THEOREM*: Herbrand (1930). If a sentence α is entailed by an FOL KB, it is entailed by a finite subset of the propositionalized KB
- *IDEA*: For *n* = 0 to ∞ do
  - create a propositional KB by instantiating with depth-*n* terms
  - see if α is entailed by this KB
- *PROBLEM*: works if α is entailed, does not halt if α is not entailed
- THEOREM: Turing (1936), Church (1936) Entailment for FOL is semi decidable
  - algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every non-entailed sentence.
- With *p* *k*-ary predicates and *n* constants, there are $p \cdot n^k$ instantiations!

Is there another way?

- Instead of translating the knowledge base to PL, we can make the inference rules work in FOL.
- For example, given
  - ∀x King(x) ∧ Greedy(x) ⇒ Evil(x)
  - King(John)
  - ∀y Greedy(y)

  It is intuitively clear that we can substitute {x/John,y/John} and obtain that Evil(John)

Unification

- We can make the inference if we can find a substitution such that *King(x)* and *Greedy(x)* match *King(John)* and *Greedy(y)*

  {x/John,y/John} works
- Unify(α ,β) = θ if Subst(θ, α) = Subst(θ, β)

α	β	Subst
Knows(John,x)	Knows(John,Jane)	
Knows(John,x)	Knows(y,OJ)	
Knows(John,x)	Knows(y,Mother(y))	
Knows(John,x)	Knows(x,OJ)	

Unification

- Unify(α ,β) = θ if Subst(θ, α) = Subst(θ, β)

α	β	Subst
Knows(John,x)	Knows(John,Jane)	{x/Jane}
Knows(John,x)	Knows(y,OJ)	
Knows(John,x)	Knows(y,Mother(y))	
Knows(John,x)	Knows(x,OJ)	

Unification

- Unify(α ,β) = θ if Subst(θ, α) = Subst(θ, β)

α	β	Subst
Knows(John,x)	Knows(John,Jane)	{x/Jane}
Knows(John,x)	Knows(y,OJ)	{x/OJ,y/John}
Knows(John,x)	Knows(y,Mother(y))	
Knows(John,x)	Knows(x,OJ)	

Unification

- Unify(α ,β) = θ if Subst(θ, α) = Subst(θ, β)

α	β	Subst
Knows(John,x)	Knows(John,Jane)	{x/Jane}
Knows(John,x)	Knows(y,OJ)	{x/OJ,y/John}
Knows(John,x)	Knows(y,Mother(y))	{y/John,x/Mother(John)}}
Knows(John,x)	Knows(x,OJ)	

Unification

- Unify(α ,β) = θ if Subst(θ, α) = Subst(θ, β)

α	β	Subst
Knows(John,x)	Knows(John,Jane)	{x/Jane}
Knows(John,x)	Knows(y,OJ)	{x/OJ,y/John}
Knows(John,x)	Knows(y,Mother(y))	{y/John,x/Mother(John)}}
Knows(John,x)	Knows(x,OJ)	{fail}

Unification

- Unifiers of *Knows(John,x)* and *Knows(y,z)* are {y/John, x/z } or {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.
  MGU = { y/John, x/z }

The unification algorithm

function UNIFY(x, y, θ) **returns** a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound
    y, a variable, constant, list, or compound
    θ, the substitution built up so far

  if θ = failure **then return** failure
  else if $x = y$ **then return** θ
  else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) **and** COMPOUND?(y) **then**
    return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
  else if LIST?(x) **and** LIST?(y) **then**
    return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
  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 $\{var/val\} \in \theta$ **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** failure
  else return add $\{var/x\}$ to θ

Generalized Modus Ponens (GMP)

Suppose that Subst(θ, p_i') = Subst(θ, p_i) *for all i then:*

$$\frac{p_1',\ p_2',\ \ldots,\ p_n',\ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{Subst}(\theta, q)}$$

p_1' is *King*(*John*) p_1 is *King*(*x*)
p_2' is *Greedy*(*y*) p_2 is *Greedy*(*x*)
θ is {x/John,y/John} q is *Evil*(*x*)
Subst(θ, q) is *Evil*(*John*)

- All variables assumed universally quantified.

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.
- Prove that Col. West is a criminal

Example

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

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles, i.e., ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1*) ∧ Missile(*M_1*)*

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles, i.e., ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) ∧ Missile(M_1)
… all of its missiles were sold to it by Colonel West

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles, i.e., ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) ∧ Missile(M_1)
… all of its missiles were sold to it by Colonel West
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles, i.e., ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) ∧ Missile(M_1)
… all of its missiles were sold to it by Colonel West
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missiles are weapons:

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles, i.e., ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) ∧ Missile(M_1)
… all of its missiles were sold to it by Colonel West
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missiles are weapons:
Missile(x) ⇒ Weapon(x)

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles, i.e., ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) ∧ Missile(M_1)
… all of its missiles were sold to it by Colonel West
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missiles are weapons:
Missile(x) ⇒ Weapon(x)
An enemy of America counts as "hostile":

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles, i.e., ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) ∧ Missile(M_1)
… all of its missiles were sold to it by Colonel West
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missiles are weapons:
Missile(x) ⇒ Weapon(x)
An enemy of America counts as "hostile":
Enemy(x,America) ⇒ Hostile(x)

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles, i.e., ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) ∧ Missile(M_1)
… all of its missiles were sold to it by Colonel West
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missiles are weapons:
Missile(x) ⇒ Weapon(x)
An enemy of America counts as "hostile":
Enemy(x,America) ⇒ Hostile(x)
West, who is American …

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles, i.e., ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) ∧ Missile(M_1)
… all of its missiles were sold to it by Colonel West
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missiles are weapons:
Missile(x) ⇒ Weapon(x)
An enemy of America counts as "hostile":
Enemy(x,America) ⇒ Hostile(x)
West, who is American …
American(West)

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles, i.e., ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) ∧ Missile(M_1)
… all of its missiles were sold to it by Colonel West
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missiles are weapons:
Missile(x) ⇒ Weapon(x)
An enemy of America counts as "hostile":
Enemy(x,America) ⇒ Hostile(x)
West, who is American …
American(West)
The country Nono, an enemy of America …

Example

... it is a crime for an American to sell weapons to hostile nations:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Nono … has some missiles, i.e., ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) ∧ Missile(M_1)
… all of its missiles were sold to it by Colonel West
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missiles are weapons:
Missile(x) ⇒ Weapon(x)
An enemy of America counts as "hostile":
Enemy(x,America) ⇒ Hostile(x)
West, who is American …
American(West)
The country Nono, an enemy of America …
Enemy(Nono,America)

Forward chaining algorithm

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

repeat until new is empty
  $new \leftarrow \{\}$
  for each sentence r **in** KB **do**
    $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-APART(r)
    for each θ such that $(p_1 \wedge \ldots \wedge p_n)\theta = (p'_1 \wedge \ldots \wedge p'_n)\theta$
        for some $p'_1, \ldots, p'_n$ in KB
      $q' \leftarrow$ SUBST(θ, q)
      if q' is not a renaming of a sentence already in KB or new **then do**
        add q' to new
        $\phi \leftarrow$ UNIFY(q', α)
        if ϕ is not *fail* **then return** ϕ
  add new to KB
return *false*

rename variables so there are no collisions

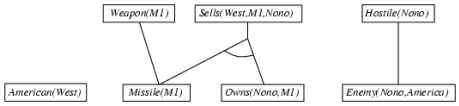
Forward chaining example

- *American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)*
- *Owns(Nono,M_1) ∧ Missile(M_1)*
- *Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)*
- *Missile(x) ⇒ Weapon(x)*
- *Enemy(x,America) ⇒ Hostile(x)*
- *American(West)*
- *Enemy(Nono,America)*

American(West) *Missile(M1)* *Owns(Nono,M1)* *Enemy(Nono,America)*

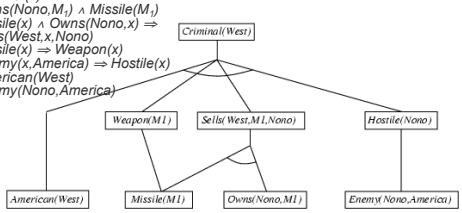
Forward chaining example

- American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
- Owns(Nono,M_1) ∧ Missile(M_1)
- Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
- Missile(x) ⇒ Weapon(x)
- Enemy(x,America) ⇒ Hostile(x)
- American(West)
- Enemy(Nono,America)



Forward chaining example

- American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
- Owns(Nono,M_1) ∧ Missile(M_1)
- Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
- Missile(x) ⇒ Weapon(x)
- Enemy(x,America) ⇒ Hostile(x)
- American(West)
- Enemy(Nono,America)



Forward chaining for FOL

- Sound and complete for first-order definite clauses.
- *Datalog* = first-order definite clauses with *no functions* (e.g. crime KB)
  - FC terminates for Datalog in finite number of iterations
- May not terminate in general definite clauses with functions if sentence is not entailed
  - This is unavoidable: entailment with definite clauses is semidecidable

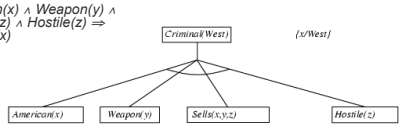
Backward chaining

- As in propositional, can work backwards from the goal
  - Requires a generator that tries multiple bindings
  - Depth-first recursive proof search
- Widely used for logic programming: problem solving by inference.
  - Example: Prolog

Backward chaining example

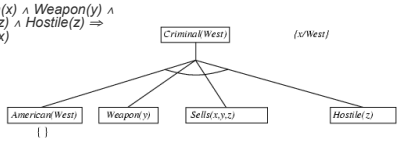
Backward chaining example

American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)

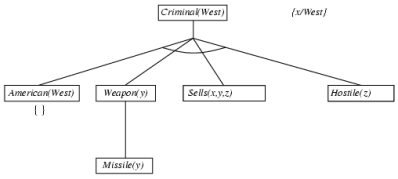


Backward chaining example

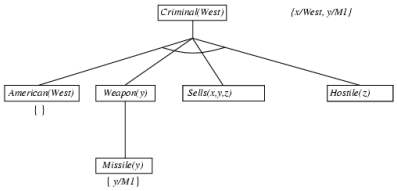
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)



Backward chaining example

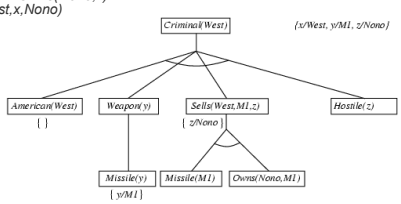


Backward chaining example

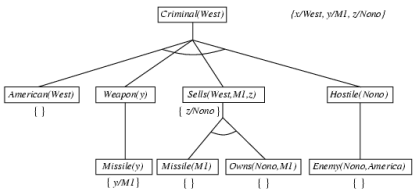


Backward chaining example

Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)



Backward chaining example



Backward chaining algorithm

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 (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(*KB*, *goal*) **do**
    (*lhs*, *rhs*) ← STANDARDIZE-VARIABLES((*lhs*, *rhs*))
    for each *θ'* **in** FOL-BC-AND(KB, lhs, UNIFY(*rhs*, *goal*, *θ*)) **do**
      yield *θ'*

---

generator FOL-BC-AND(*KB*, *goal*, *θ*) **yields** *a substitution*
  if *θ* = *failure* **then return**
  else if *length(goals)* = *0* **then yield** *θ*
  else do
    first,rest ← FIRST(*goals*), REST(*goals*)
    for each *θ'* **in** FOL-BC-OR(*KB*, *SUBST(θ, first)*, *θ*) **do**
      for each *θ"* **in** FOL-BC-AND(*KB*, *SUBST(θ', rest)*, *θ'*) **do**
        yield *θ"*

There may be multiple relevant substitutions, so the functions are generators

Properties of backward chaining

Depth-first recursive proof search

- space is linear in size of proof.
- Incomplete due to infinite loops
  - Fixable
- Inefficient due to repeated subgoals
  - Fixable by caching previous results (extra space!!)
- Widely used for logic programming: problem solving by inference.
  - Example: Prolog

Logic programming: Prolog

- BASIS: backward chaining with Horn clauses + bells & whistles
- Program = set of clauses of the form

  head :- literal$_1$, … literal$_n$.

  criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).

Prolog is a 'declarative' language

- Clauses are statements about what is true about a problem, instead of instructions how to accomplish the solution.
- The Prolog system uses the clauses to work out how to accomplish the solution by searching through the space of possible solutions.

Example

- Prolog program consists of facts and rules.

```
animal(lion).
animal(sparrow).
hasfeathers(sparrow).
bird(X) :- animal(X), hasfeathers(X).
```

- "Run" by asking questions or queries. Or (using logic terminology) by setting a *goal* for Prolog to try to prove:

```
?- bird(sparrow).
yes
```

- Or to find a value of a variable that makes it true:

```
?- bird(What).
What = sparrow
```


Example

- Appending two lists to produce a third:

```
append([],Y,Y).
append([X|L],Y,[X|Z]) :- append(L,Y,Z).
```

- query: `append(A,B,[1,2]).`
- answers:

```
A=[]    B=[1,2]
A=[1]   B=[2]
A=[1,2] B=[]
```


Example

ancestor(X, Y) :- parent(X, Y).

(X is an ancestor of Y if X is a parent of X.)

ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

(X is an ancestor of Y if X is a parent of an ancestor of Y).

Permutations

- Permutation(X, Y) is true whenever Y is a permutation of X.

```
?-permuation([a,b,c], P).
P = [a,b,c];
P = [a,c,b];
P = [b,a,c];
```

Two cases:

- The only permutation of the empty list is the empty list.
- If the first list is not empty, the it has the form [X|L], and a permutation of such a list can be constructed by first permuting L and then inserting X at any position into the permuted list.

```
permutation([], []).
permutation([X|L], P) :-
        permutation(L, L1), insert(X, L1, P).
```


Resolution

- Full first-order version:

$$\frac{l_1 \vee \cdots \vee l_k, \qquad m_1 \vee \cdots \vee m_n}{Subst(\theta, l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where $\theta = Unify(l_i, \neg m_j)$

- The two clauses are assumed to be standardized apart so that they share no variables.
- For example,

$$\frac{\neg Rich(x) \vee Unhappy(x), \qquad Rich(Ken)}{Unhappy(Ken)}$$

with $\theta = \{x/Ken\}$

- Apply resolution steps to $CNF(KB \wedge \neg\alpha)$; complete for FOL

Conversion to CNF

- Need to convert KB to CNF, eliminating existential quantifiers
- Consider the sentence "There is someone who is loved by everyone":

  ∃y∀x Loves(x, y)

  Let's name that someone using a constant that does not appear elsewhere in the KB (Skolem constant):

  ∀x Loves(x, Sk1)
- Let's try now the sentence "Everyone is loved by someone"

  ∀y∃x Loves(x, y)

  Consider the following Skolemization:

  ∀y Loves(SK2, y)

Conversion to CNF

- Need to convert KB to CNF, eliminating existential quantifiers
- Consider the sentence "There is someone who is loved by everyone":

  ∃y∀x Loves(x, y)

  Let's name that someone using a constant that does not appear elsewhere in the KB (Skolem constant):

  ∀x Loves(x, Sk1)
- Let's try now the sentence "Everyone is loved by someone"

  ∀y∃x Loves(x, y)

  Consider the following Skolemization:

  ∀y Loves(SK2, y)
- This doesn't work, but skolemization using a function does:

  ∀y Loves(SK2(y), y)

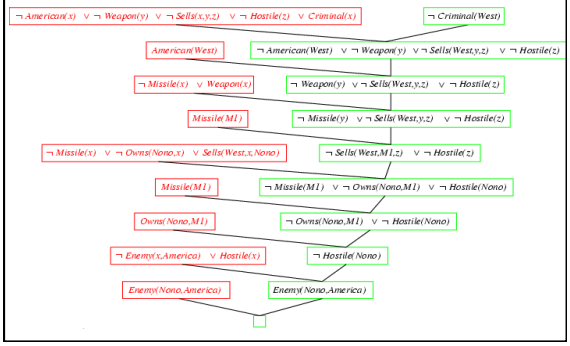
Conversion to CNF

- Everyone who loves all animals is loved by someone:
  ∀x ([∀y *Animal*(y) ⇒ *Loves*(x,y)] ⇒ [∃y *Loves*(y,x)])
- Eliminate implications (and biconditionals)
  ∀x ([¬(∀y ¬*Animal*(y) ∨ *Loves*(x,y))] ∨ [∃y *Loves*(y,x)])
- Move ¬ inwards: ¬∀x p ≡ ∃x ¬p, ¬ ∃x p ≡ ∀x ¬p
  ∀x [∃y ¬(¬*Animal*(y) ∨ *Loves*(x,y))] ∨ [∃y *Loves*(y,x)]
  ∀x [∃y ¬¬*Animal*(y) ∧ ¬*Loves*(x,y)] ∨ [∃y *Loves*(y,x)]
  ∀x [∃y *Animal*(y) ∧ ¬*Loves*(x,y)] ∨ [∃y *Loves*(y,x)]

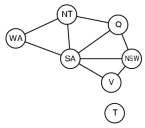
Conversion to CNF

- Standardize variables: each quantifier should use a different one:
  ∀x [∃y *Animal*(y) ∧ ¬*Loves*(x,y)] ∨ [∃z *Loves*(z,x)]
- *Skolemize*: Each existential variable is replaced by a Skolem function of the enclosing universally quantified variables:
  ∀x [*Animal*(*F*(x)) ∧ ¬*Loves*(x,*F*(x))] ∨ *Loves*(*G*(x),x)
- Drop universal quantifiers:
  [*Animal*(*F*(x)) ∧ ¬*Loves*(x,*F*(x))] ∨ *Loves*(*G*(x),x)
- Distribute ∨ over ∧ :
  [*Animal*(*F*(x)) ∨ *Loves*(*G*(x),x)] ∧ [¬*Loves*(x,*F*(x)) ∨ *Loves*(*G*(x),x)]

Resolution proof



Connection with CSP



Diff(wa,nt) ∧ *Diff(wa,sa)* ∧ *Diff(nt,q)* ∧ *Diff(nt,sa)* ∧ *Diff(q,nsw)* ∧ *Diff(q,sa)* ∧ *Diff(nsw,v)* ∧ *Diff(nsw,sa)* ∧ *Diff(v,sa)* ⇒ *Colorable()*

Diff(Red,Blue) *Diff (Red,Green)*
Diff(Green,Red) *Diff(Green,Blue)*
Diff(Blue,Red) *Diff(Blue,Green)*

- *Colorable*() is inferred iff the CSP has a solution
- Hardness of inference problem related to the difficulty of the corresponding CSP.

Theorem provers

- Theorem prover – a system that does full first order logic inference (using resolution)
- Uses:
  - Prove mathematical theorems (known successes!)
  - Hardware/software verification
    - Verify that circuit/software produces correct output for all possible inputs (RSA algorithm was verified this way)