First Order Logic: Inference

Forest Agostinelli
University of South Carolina

Topics Covered in This Class

- **Part 1: Search**
  - Pathfinding
    - Uninformed search
    - Informed search
  - Adversarial search
  - Optimization
    - Local search
    - Constraint satisfaction
- **Part 2: Knowledge Representation and Reasoning**
  - Propositional logic
  - First-order logic
  - Prolog
- **Part 3: Knowledge Representation and Reasoning Under Uncertainty**
  - Probability
  - Bayesian networks
- **Part 4: Machine Learning**
  - Supervised learning
    - Inductive logic programming
    - Linear models
    - Deep neural networks
    - PyTorch
  - Reinforcement learning
    - Markov decision processes
    - Dynamic programming
    - Model-free RL
  - Unsupervised learning
    - Clustering
    - Autoencoders

Outline

- Inference in FOL using modus ponens
- Unification
- Inference in FOL using Unification and Resolution
- Examples

Modus Ponens Example

- $\forall x\ Tomato(x) \land Red(x) \rightarrow Ripe\ (x)$
- $Tomato(Fruit1)$
- $Red(Fruit1)$
- $Green(Fruit2)$

- Ripe(Fruit1)?
  - In the first sentence, we make the substitution $\theta = \{x/Fruit1\}$
  - $Tomato(Fruit1) \land Red(Fruit1) \rightarrow Ripe\ (Fruit1)$
  - Using the other two sentences in our KB and modus ponens, we can show that Fruit1 is ripe

Modus Ponens Example

- $\forall x\; Tomato(x) \land Red(x) \rightarrow Ripe\ (x)$
- $\forall y\; Tomato(y)$
- $Red(Fruit1)$
- $Green(Fruit2)$

- Ripe(Fruit1)?
  - In the first sentence, we make the substitution $\theta = \{x/Fruit1\}$
    - $Tomato(Fruit1) \land Red(Fruit1) \rightarrow Ripe\ (Fruit1)$
  - In the second sentence, we make the substitution $\theta = \{y/Fruit1\}$
    - $Tomato(Fruit1)$

Resolution in FOL

- For any sentences α and β $\alpha \vDash \beta$ iff the sentence $\alpha \rightarrow \beta$ is valid
  - $\alpha \vDash \beta$ iff $\alpha \wedge \neg\beta$ is unsatisfiable
    - Proof by contradiction
  - To show that $KB \vDash \alpha$ we show $KB \wedge \neg\alpha$ is unsatisfiable
- Resolution is refutation complete in propositional logic and FOL
  - If a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction
- We know how to do resolution in propositional logic
- We can convert FOL sentences to CNF (removing quantifiers) and then use unification to do resolution

Completeness

- If a knowledge base has a function symbol, say Friend, the space of possible models is infinite
  - I.e. Friend(Friend(Friend(Friend(X))))
- However, if a sentence is entailed by the knowledge base, then a proof can be found in a finite amount of time
- Therefore, inference in first-order logic is complete
  - Any entailed sentence can be proved

Decidability

- If a sentence is not entailed by the knowledge base, resolution may or may not halt
  - In other words, it could run forever
- Therefore, proof by resolution in first-order logic is semi-decidable
  - Unlike in propositional logic where it is fully decidable because it can be posed as a SAT problem

Unification

- Since we have variables, we need to find substitutions to make different logical expressions look identical
- The Unify algorithm takes two sentences and returns a unifier for them, if one exists
  - $Unify(p, q) = \theta$ where $Subst(\theta, p) = Subst(\theta, q)$

Unification Example

- Query is $Knows(John, x)$
- Standardizing variables apart in different clauses will lead to success in the last example
  - E.g. $Knows(z, OJ)$

p	q	θ
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}

Most General Unifier

- Unify(Knows(John,x), Knows(y,z))
  - θ = {x/John, y/John, z/John}
  - -or-
  - θ = {y/John, x/z}
- The second one is more general
  - Does not unnecessarily rule out other possibilities
  - John could know some other person, not just himself
- In unification, we want to find the most general unifier

Quick Quiz

UNIFY(Knows(John, x), Knows(John, Jane))

UNIFY(Knows(John, x), Knows(y, Jane))

UNIFY(Knows(y, x), Knows(John, Jane))

UNIFY(Knows(John, x), Knows(y, Father (y)))

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

UNIFY(Knows(John, F(x)), Knows(y, G(z)))

UNIFY(Knows(John, F(x)), Knows(y, F(G(y))))

Quick Quiz

UNIFY(Knows(John, x), Knows(John, Jane))	{ x / Jane }
UNIFY(Knows(John, x), Knows(y, Jane))	{ x / Jane, y / John }
UNIFY(Knows(y, x), Knows(John, Jane))	{ x / Jane, y / John }
UNIFY(Knows(John, x), Knows(y, Father (y)))	{ y / John, x / Father (John) }
UNIFY(Knows(John, F(x)), Knows(y, F(F(z))))	{ y / John, x / F (z) }
UNIFY(Knows(John, F(x)), Knows(y, G(z)))	None
UNIFY(Knows(John, F(x)), Knows(y, F(G(y))))	{ y / John, x / G (John) }

Unification

```
function UNIFY(x, y, θ=empty) returns a substitution to make x and y identical, or failure
   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

function UNIFY-VAR(var, x, θ) returns a substitution
   if {var/val} ∈ θ for some val then return UNIFY(val, x, θ)
   else if {x/val} ∈ θ for some val then return UNIFY(var, val, θ)
   else if OCCUR-CHECK?(var, x) then return failure
   else return add {var/x} to θ
```

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A, B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A, B).

Unification

function UNIFY(x, y, θ=*empty*) **returns** a substitution to make x and y identical, or *failure*
  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*

Check for failure or success

function UNIFY-VAR(var, x, θ) **returns** a substitution
  if $\{var/val\} \in \theta$ for some val **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ for some val **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** *failure*
  else return add $\{var/x\}$ to θ

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A, B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A, B).

Unification

```
function UNIFY(x, y, θ=empty) returns a substitution to make x and y identical, or failure
  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

function UNIFY-VAR(var, x, θ) returns a substitution
  if {var/val} ∈ θ for some val then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ for some val then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

If we can unify a variable, then do so

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A, B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A, B).

Unification

function UNIFY(x, y, θ=*empty*) **returns** a substitution to make x and y identical, or *failure*
  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*

If a predicate or function then unify the arguments

function UNIFY-VAR(var, x, θ) **returns** a substitution
  if $\{var/val\} \in \theta$ for some val **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ for some val **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** *failure*
  else return add $\{var/x\}$ to θ

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A, B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A, B).

Unification

```
function UNIFY(x, y, θ=empty) returns a substitution to make x and y identical, or failure
   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

function UNIFY-VAR(var, x, θ) returns a substitution
   if {var/val} ∈ θ for some val then return UNIFY(val, x, θ)
   else if {x/val} ∈ θ for some val then return UNIFY(var, val, θ)
   else if OCCUR-CHECK?(var, x) then return failure
   else return add {var/x} to θ
```

If unifying arguments, unify the first arguments, then unify the remaining arguments

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A, B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A, B).

Unification

function UNIFY(x, y, θ=*empty*) **returns** a substitution to make x and y identical, or *failure*
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*

Otherwise, return failure

function UNIFY-VAR(var, x, θ) **returns** a substitution
if $\{var/val\} \in \theta$ for some val **then return** UNIFY(val, x, θ)
else if $\{x/val\} \in \theta$ for some val **then return** UNIFY(var, val, θ)
else if OCCUR-CHECK?(var, x) **then return** *failure*
else return add $\{var/x\}$ to θ

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A, B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A, B).

Unification

function UNIFY(x, y, θ=*empty*) **returns** a substitution to make x and y identical, or *failure*
  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*

function UNIFY-VAR(var, x, θ) **returns** a substitution
  if $\{var/val\} \in \theta$ for some val **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ for some val **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** *failure*
  else return add $\{var/x\}$ to θ

If we have already substituted val for var, then try and unify val and x

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A, B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A, B).

Unification

```
function UNIFY(x, y, θ=empty) returns a substitution to make x and y identical, or failure
  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

function UNIFY-VAR(var, x, θ) returns a substitution
  if {var/val} ∈ θ for some val then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ for some val then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

If we have already substituted val for x, then try to unify var and val

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A, B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A, B).

Unification

function UNIFY(x, y, θ=*empty*) **returns** a substitution to make x and y identical, or *failure*
  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*

function UNIFY-VAR(var, x, θ) **returns** a substitution
  if $\{var/val\} \in \theta$ for some val **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ for some val **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** *failure*
  else return add $\{var/x\}$ to θ

If var occurs anywhere within x then no substitution will succeed

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A, B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A, B).

Unification

function UNIFY(x, y, θ=*empty*) **returns** a substitution to make x and y identical, or *failure*
  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*

function UNIFY-VAR(var, x, θ) **returns** a substitution
  if $\{var/val\} \in \theta$ for some val **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ for some val **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** *failure*
  else return add $\{var/x\}$ to θ

Substitute x for var

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A, B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A, B).

Proof by Contradiction Using Resolution

- Given some knowledge base KB and some query α
- Negate α
- Convert KB and $\neg\alpha$ to CNF
- Perform resolution until there is a contradiction or until there are no more clauses to resolve

Conjunctive Normal Form

- Eliminate biconditionals and implications
- Move $\neg$ inwards
  - DeMorgan's Rule
- Standardize variables
  - If there are different variables with the same name in a sentence, give them different names
- Skolemize
  - Remove existential quantifiers by replacing their variables with new objects
  - $\exists x\ Loves(x, Friend(x))$ becomes $Loves(P, Friend(P))$
  - $\forall x \exists y\ Loves(x, y)$ becomes $\forall x\ Loves(x, G(x))$
    - This can be different for each x
- Drop universal quantifiers
- Distribute over $\vee$ and $\wedge$

Simple Proof by Contradiction Using Resolution

- KB
  - $Fruit(Tomato)$
  - $Vegetable(Carrot)$
- Query
  - $\exists x\ Fruit(x)$
- Negate query
  - $\neg\exists x\ Fruit(x) \equiv \forall x\ \neg Fruit(x)$
- Drop universal quantifiers
  - $\neg Fruit(x)$
- Contradiction
  - Resolve $\neg Fruit(x)$ and $Fruit(Tomato)$ with {x/Tomato}

Simple Proof by Contradiction Using Resolution

- KB
  - $\exists x\ Fruit(x)$
- Query
  - $\exists x\ Fruit(x)$
- Negate query
  - $\neg\exists x\ Fruit(x) \equiv \forall x\ \neg Fruit(x)$
- Skolemize
  - $\exists x\ Fruit(x)$ becomes $Fruit(F)$
- Drop universal quantifiers
  - $\neg Fruit(x)$
- Contradiction
  - Resolve $\neg Fruit(x)$ and $Fruit(F)$ with {x/F}

Resolution Example

- KB
  - "Everyone who loves all animals is loved by someone"
    - $\forall x \left(\forall y Animal(y) \rightarrow Loves(x, y)\right) \rightarrow \exists y\ Loves(y, x)$
  - "Jack loves all animals"
    - $\forall x\ Animal(x) \rightarrow Loves(Jack, x)$
- Query
  - "Does someone love Jack?"
  - $\exists x\ Loves(x, Jack)$
- Negate query
  - $\neg \exists x\ Loves(x, Jack)$
  - $\forall x\ \neg Loves(x, Jack)$

Eliminate Implications

- $\forall x\ \left(\forall y Animal(y) \rightarrow Loves(x,y)\right) \rightarrow \exists y\ Loves(y,x)$
  - $\forall x\ \left(\forall y\ \neg Animal(y) \vee Loves(x,y)\right) \rightarrow \exists y\ Loves(y,x)$
  - $\forall x\ \neg\left(\forall y\ \neg Animal(y) \vee Loves(x,y)\right) \vee \exists y\ Loves(y,x)$
- $\forall x\ Animal(x) \rightarrow Loves(Jack,x)$
  - $\forall x\ \neg Animal(x) \vee Loves(Jack,x)$

Move ¬ Inwards

- $\forall x\ \neg\left(\forall y\ \neg Animal(y) \lor Loves(x,y)\right) \lor \exists y\ Loves(y,x)$
  - $\forall x\ \left(\exists y\ Animal(y) \land \neg Loves(x,y)\right) \lor \exists y\ Loves(y,x)$

Standardize Variables

- $\forall x\ \left(\exists y\ Animal(y) \land \neg Loves(x,y)\right) \lor \exists y\ Loves(y,x)$
    - $\forall x\ \left(\exists y\ Animal(y) \land \neg Loves(x,y)\right) \lor \exists z\ Loves(z,x)$

Skolemization

- $\forall x \left(\exists y\ Animal(y) \land \neg Loves(x, y) \right) \lor \exists z\ Loves(z, x)$
  - $\forall x \left(Animal(F(x)) \land \neg Loves(x, F(x)) \right) \lor Loves(G(x), x)$

Drop Universal Quantifiers

- $\forall x \left(Animal(F(x)) \land \neg Loves(x, F(x))\right) \lor Loves(G(x), x)$
  - $\left(Animal(F(x)) \land \neg Loves(x, F(x))\right) \lor Loves(G(x), x)$
- $\forall x \neg Animal(x) \lor Loves(Jack, x)$
  - $\neg Animal(x) \lor Loves(Jack, x)$

Distribute over ∨ and ∧

- $\left(Animal(F(x)) \wedge \neg Loves\left(x, F(x)\right)\right) \vee Loves(G(x), x)$
  - $\left(Animal\left(F(x)\right) \vee Loves(G(x), x)\right) \wedge \left(\neg Loves\left(x, F(x)\right) \vee Loves(G(x), x)\right)$

Proof

1. $Animal(F(x)) \vee Loves(G(x), x))$
2. $\neg Loves(x, F(x)) \vee Loves(G(x), x))$
3. $\neg Animal(x) \vee Loves(Jack, x)$
4. $\neg Loves(x, Jack)$ (negation of what we want to prove)
5. $\neg Loves(Jack, F(Jack))$
    1. Resolve 4 and 2 {x/Jack, y/G(Jack)}. Must standardize variables apart: change x in 4 to y.
6. $Animal(F(Jack))$
    1. Resolve 4 and 1 {x/Jack, y/G(Jack)}. Must standardize variables apart: change x in 4 to y.
7. $Loves(Jack, F(Jack))$
    1. Resolve 6 and 3 {x/F(Jack)}
8. Contradiction!
    1. Resolve 7 and 5

Resolution Example #2

- KB
  - "Everyone who loves some animal is loved by someone"
    - $\forall x \left(\exists y Animal(y) \wedge Loves(x,y)\right) \rightarrow \exists y\ Loves(y,x)$
  - "Jack loves all animals"
    - $\forall x\ Animal(x) \rightarrow Loves(Jack, x)$
  - "There exists an animal" (what happens if we remove this ?)
    - $\exists y Animal(y)$
- Query
  - "Does someone love Jack?"
  - $\exists x\ Loves(x, Jack)$
- Negate query
  - $\forall x\ \neg Loves(x, Jack)$

Conjunctive Normal Form

- $\forall x \left(\exists y Animal(y) \land Loves(x,y)\right) \rightarrow \exists y\ Loves(y,x)$
  - $\forall x\ \neg\left(\exists y Animal(y) \land Loves(x,y)\right) \lor \exists y\ Loves(y,x)$
  - $\forall x \left(\forall y \neg Animal(y) \lor \neg Loves(x,y)\right) \lor \exists y\ Loves(y,x)$
  - $\forall x \left(\forall y \neg Animal(y) \lor \neg Loves(x,y)\right) \lor \exists z\ Loves(z,x)$
  - $\forall x \left(\forall y \neg Animal(y) \lor \neg Loves(x,y)\right) \lor Loves(G(x),x)$
  - $\neg Animal(y) \lor \neg Loves(x,y) \lor Loves(G(x),x)$
- $\exists y Animal(y)$
  - $Animal(A)$

Proof

1. $\neg Animal(y) \lor \neg Loves(x, y) \lor Loves(G(x), x)$
2. $\neg Animal(x) \lor Loves(Jack, x)$
3. $Animal(A)$
4. $\neg Loves(x, Jack)$ (negation of what we want to prove)
5. $Loves(Jack, A)$
   - Resolve 2 and 3 {x/A}
6. $\neg Loves(x, A) \lor Loves(G(x), x)$
   - Resolve 1 and 3 {x/A}
7. $Loves(G(Jack), Jack)$
   - Resolve 5 and 6 {x/Jack}
8. Contradiction!
   - Resolve 4 and 7 {x/G(Jack)}

Did Curiosity Kill the Cat?

- Anyone who loves all animals is loved by someone
- Anyone who kills an animal is loved by no one
- Jack loves all animals
- Jack or Curiosity killed Tuna
- Tuna is a cat
- All cats are animals

- Did Curiosity kill Tuna?

Proof By Contradiction (in English)

- Assume Curiosity did not kill Tuna
- If Curiosity did not kill Tuna then Jack must have killed Tuna
- Tuna is a cat and therefore an animal
- Jack killed an animal (Tuna); therefore, Jack is loved by no one
- However, Jack loves all animals; therefore, someone must love Jack
- The previous two sentences contradict each other
- Therefore, Curiosity must have killed the cat

Knowledge Base

- $\forall x \left(\forall y Animal(y) \rightarrow Loves(x, y)\right) \rightarrow \exists y\ Loves(y, x)$
- $\forall x \left(\exists z\ Animal(z) \land Kills(x, z)\right) \rightarrow \forall y\ \neg Loves(y, x)$
- $\forall x\ Animal(x) \rightarrow Loves(Jack, x)$
- $Kills(Jack, Tuna) \lor Kills(Curiosity, Tuna)$
- $Cat(Tuna)$
- $\forall x\ Cat(x) \rightarrow Animal(x)$
- $\neg Kills(Curiosity, Tuna)$

Convert to CNF

- $\forall x \left(\exists z\ Animal(z) \land Kills(x,z)\right) \rightarrow \forall y\ \neg Loves(y,x)$
- $\forall x\ Cat(x) \rightarrow Animal(x)$

- Eliminate implications
- Move $\neg$ inwards
- Standardize variables
- Skolemize
- Drop universal quantifiers
- Distribute over $\lor$ and $\land$

CNF

First-Order Logic (after converting sentences to CNF):

1) $Animal(F(x)) \vee Loves(G(x), x)$
2) $\neg Loves(x, F(x)) \vee Loves(G(x), x)$
3) $\neg Animal(z) \vee \neg Kills(x, z) \vee \neg Loves(y, x)$
4) $\neg Animal(x) \vee Loves(Jack, x)$
5) $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$
6) $Cat(Tuna)$
7) $\neg Cat(x) \vee Animal(x)$

Proof

First-Order Logic (after converting sentences to CNF):

1) $Animal(F(x)) \vee Loves(G(x), x)$
2) $\neg Loves(x, F(x)) \vee Loves(G(x), x)$
3) $\neg Animal(z) \vee \neg Kills(x, z) \vee \neg Loves(y, x)$
4) $\neg Animal(x) \vee Loves(Jack, x)$
5) $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$
6) $Cat(Tuna)$
7) $\neg Cat(x) \vee Animal(x)$

8) $\neg Kills(Curiosity, Tuna)$ (The negation of what we want to prove)

9) $Kills(Jack, Tuna)$ //Resolve 5 and 8. If Curiosity did not kill Tuna then Jack must have killed Tuna.
10) Animal(Tuna) //Resolve 6 and 7 $\{x/Tuna\}$. Tuna is a cat and therefore an animal.
11) $\neg Animal(Tuna) \vee \neg Loves(y, Jack)$ //Resolve 3 and 9 $\{x/Jack, z/Tuna\}$. Either Tuna is not an animal, or nobody loves Jack.
12) $\neg Loves(y, Jack)$ //Resolve 10 and 11. Tuna is an animal; therefore, nobody loves Jack.
13) $\neg Animal(F(Jack)) \vee Loves(G(Jack), Jack)$ //Resolve 2 and 4, variables for these two sentences must be standardized apart. For this resolution, change x in sentence 4 to y. $\{x/Jack, y/F(Jack)\}$.
14) $Loves(G(Jack), Jack)$ //Resolve 1 and 13 $\{x/Jack\}$. Somebody loves Jack.
15) Contradiction! //Resolve 12 and 14 $\{y/G(Jack)\}$

Therefore, Curiosity must have killed Tuna (the cat)!

Summary

- Converting to CNF requires us to also standardize variables apart, Skolemize, and drop universal quantifiers
- When performing resolution, we must do unification
  - We want to use the most general unifier
  - We also must standardize variables apart when we are doing resolution

Next Time

- Prolog