Lecture notes for *Principles of Artificial Intelligence* (COMS 4720/5720)
Yan-Bin Jia, Iowa State University

Resolution in FOL

Outline

I. Logic programming

II. Resolution

* Figures are from the textbook site.

I. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

I. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

I. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

// $American(x) \wedge Weapon(y) \wedge Hostile(z) \wedge Sells(x, y, z) \Rightarrow Criminal(x)$

I. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

// *American*$(x) \wedge$ *Weapon*$(y) \wedge$ *Hostile*$(z) \wedge$ *Sells*$(x, y, z) \Rightarrow$ *Criminal*(x)

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

I. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

// *American*$(x) \wedge$ *Weapon*$(y) \wedge$ *Hostile*$(z) \wedge$ *Sells*$(x, y, z) \Rightarrow$ *Criminal*(x)

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

$\Leftarrow$

I. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

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

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

⇐ ∧

I. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

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

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

⇐ ∧ uppercase letters for variables

I. Logic Programming

Algorithm = Logic + Control (Robert Kowalski)

Prolog (1972) is the most widely used logic programming language.

- Rapid prototyping
- Symbolic manipulation (e.g., writing compliers, parsing natural languages)

♦ A Prolog program is a set of definite clauses.

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

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

⇐ ∧ uppercase letters for variables; end of a clause

Backward Chaining in Prolog

- Prolog recursively defines a function.

Backward Chaining in Prolog

- Prolog recursively defines a function.

Example List appending.

```
append([], Y, Y).
```

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

Backward Chaining in Prolog

- Prolog recursively defines a function.

Example List appending.

```
// appending the empty list and the list Y produces the same list Y.
append([], Y, Y).
```

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

Backward Chaining in Prolog

- Prolog recursively defines a function.

Example List appending.

```
// appending the empty list and the list Y produces the same list Y.
append([], Y, Y).
```

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

a list whose first element
is A and rest is X.

Backward Chaining in Prolog

- Prolog recursively defines a function.

Example List appending.

```
// appending the empty list and the list Y produces the same list Y.
append([], Y, Y).

// [A | Z] is the result of appending [A | X] and Y provided that Z is
// the result of appending X and Y.
append( [A|X], Y, [A|Z]) :- append(X,Y,Z).
```

a list whose first element
is A and rest is X.

Backward Chaining in Prolog

- Prolog recursively defines a function.

Example List appending.

```
// appending the empty list and the list Y produces the same list Y.
append([], Y, Y).

// [A | Z] is the result of appending [A | X] and Y provided that Z is
// the result of appending X and Y.
append( [A|X], Y, [A|Z]) :- append(X,Y,Z).
```

a list whose first element
is A and rest is X.

Describes the relations among the three arguments of append.

Query Example

(1) `append([], Y, Y).`

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

Query: `append(X, Y, [1, 2, 3])`

Query Example

(1) `append([], Y, Y).`

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

Query: `append(X, Y, [1, 2, 3])`

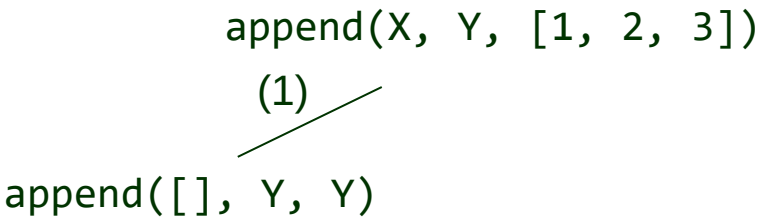
`append(X, Y, [1, 2, 3])`

Query Example

(1) `append([], Y, Y).`

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

Query: `append(X, Y, [1, 2, 3])`

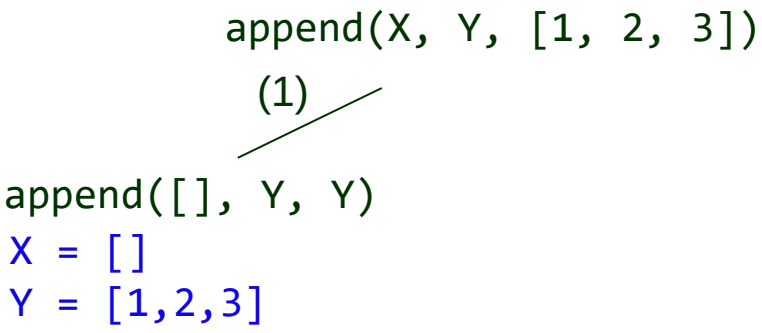


Query Example

(1) `append([], Y, Y).`

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

Query: `append(X, Y, [1, 2, 3])`

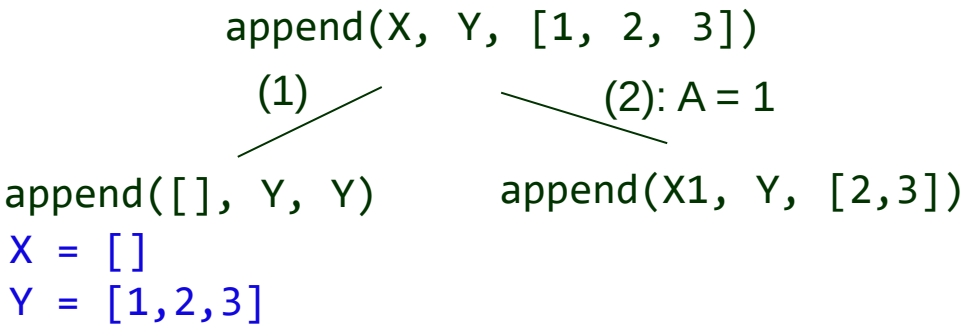


Query Example

(1) `append([], Y, Y).`

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

Query: `append(X, Y, [1, 2, 3])`

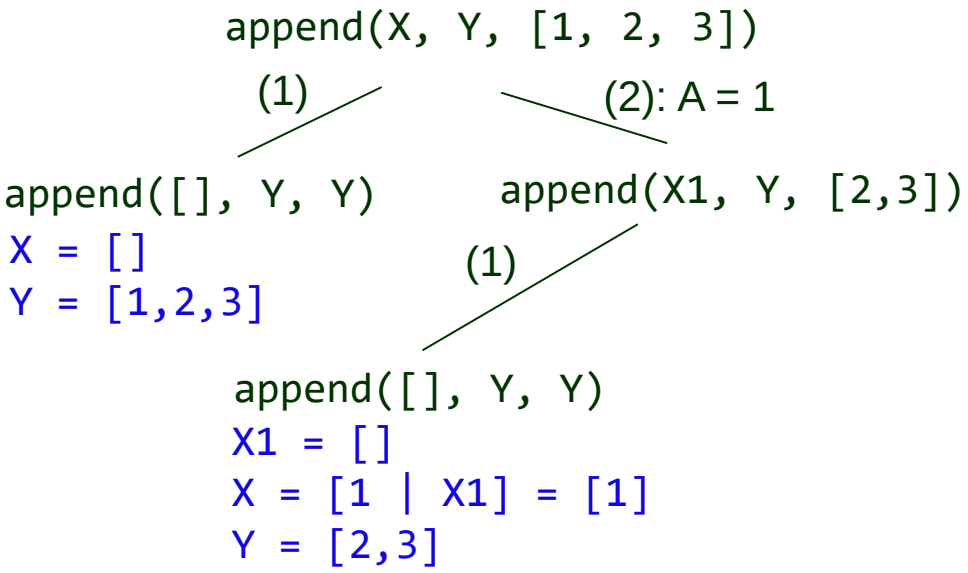


Query Example

(1) `append([], Y, Y).`

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

Query: `append(X, Y, [1, 2, 3])`

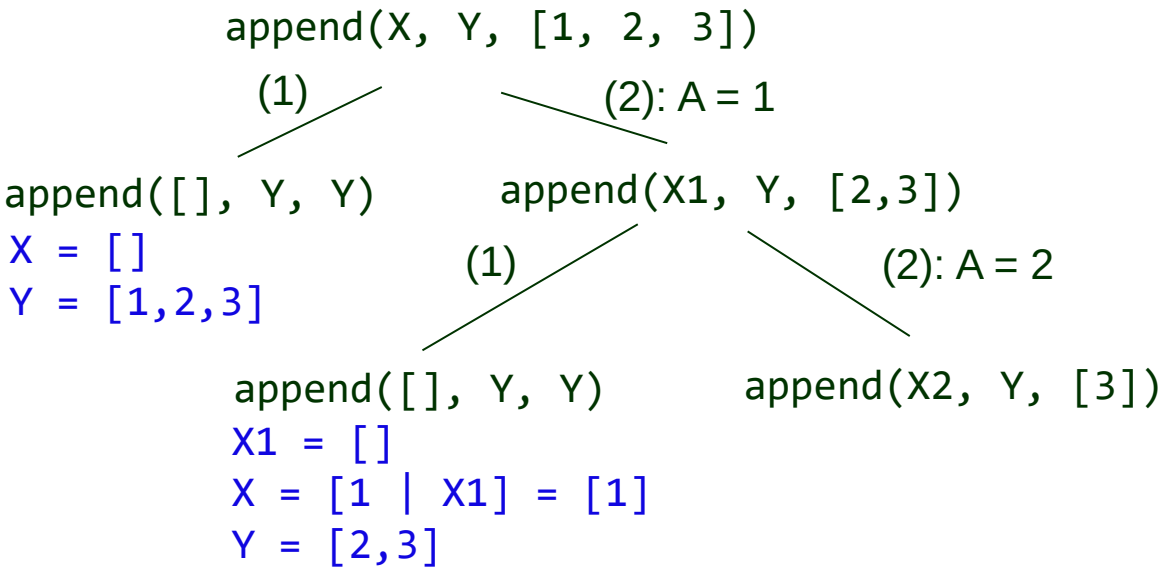


Query Example

(1) `append([], Y, Y).`

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

Query: `append(X, Y, [1, 2, 3])`

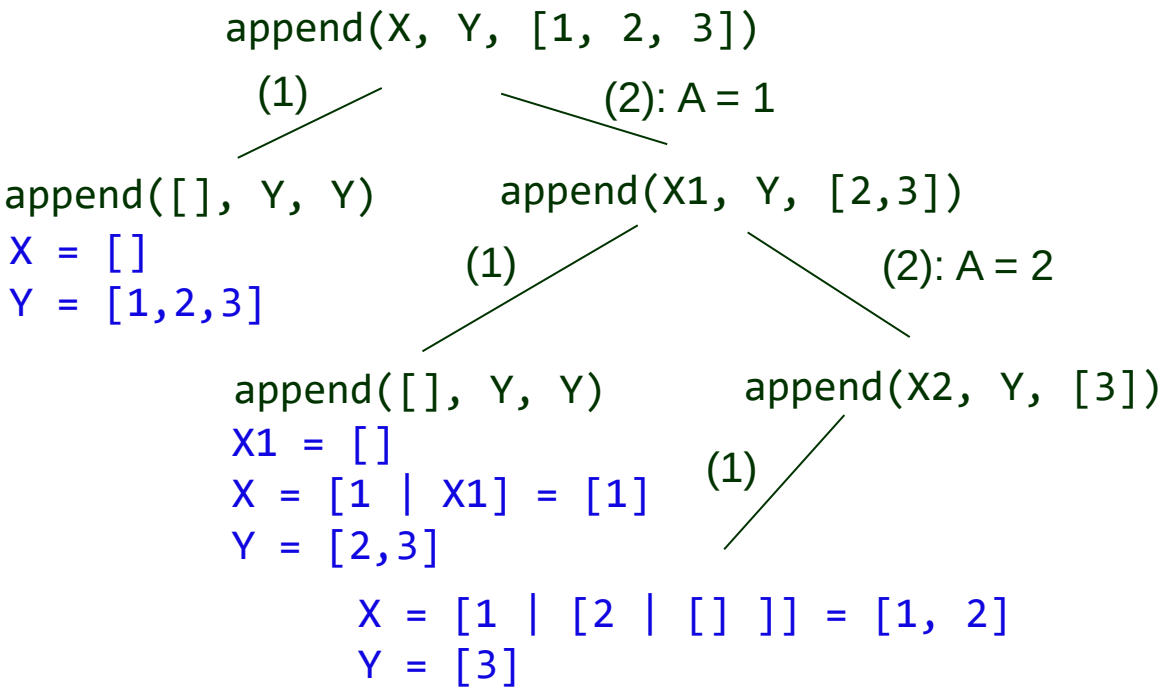


Query Example

(1) `append([], Y, Y).`

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

Query: `append(X, Y, [1, 2, 3])`

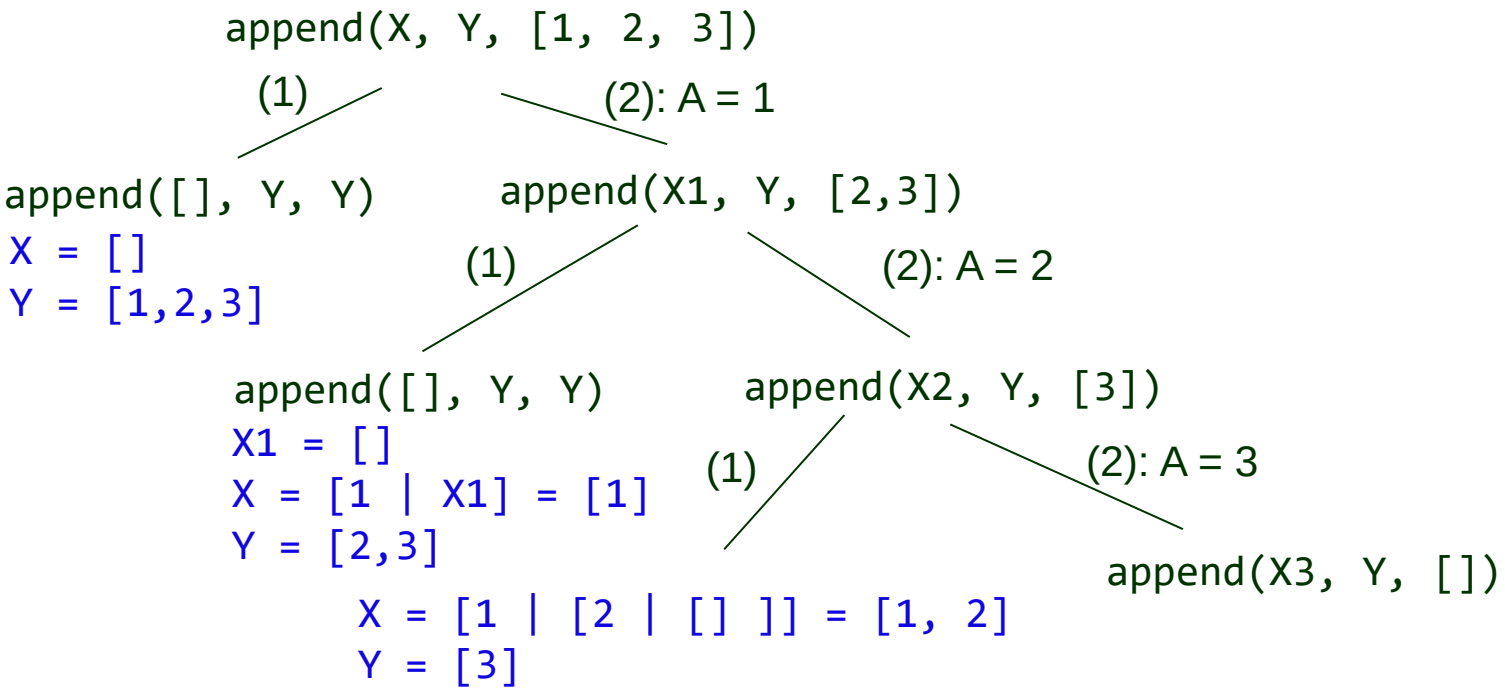


Query Example

(1) `append([], Y, Y).`

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

Query: `append(X, Y, [1, 2, 3])`

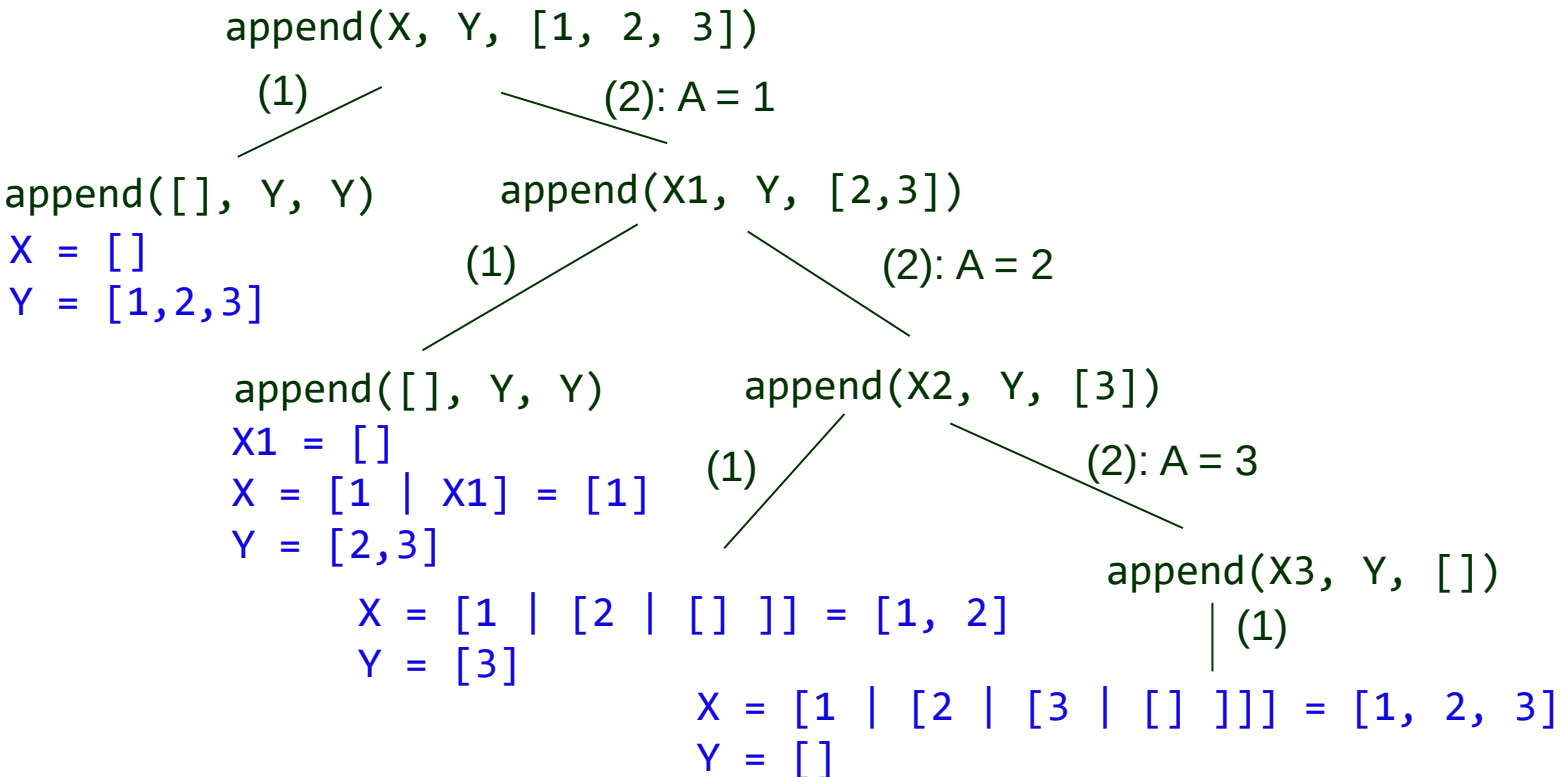


Query Example

(1) `append([], Y, Y).`

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

Query: `append(X, Y, [1, 2, 3])`



Infinite Loop

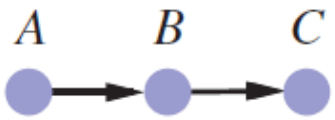
Finds if a path exists between two nodes in a directed graph.

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

Infinite Loop

Finds if a path exists between two nodes in a directed graph.

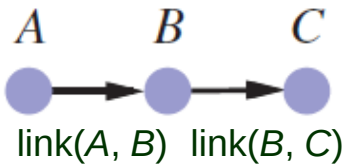
```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```



Infinite Loop

Finds if a path exists between two nodes in a directed graph.

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

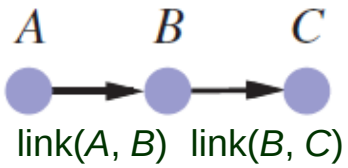


Infinite Loop

Finds if a path exists between two nodes in a directed graph.

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

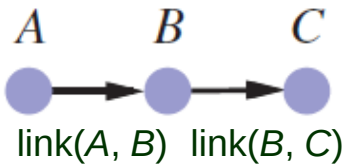
Query `path(a, c)`



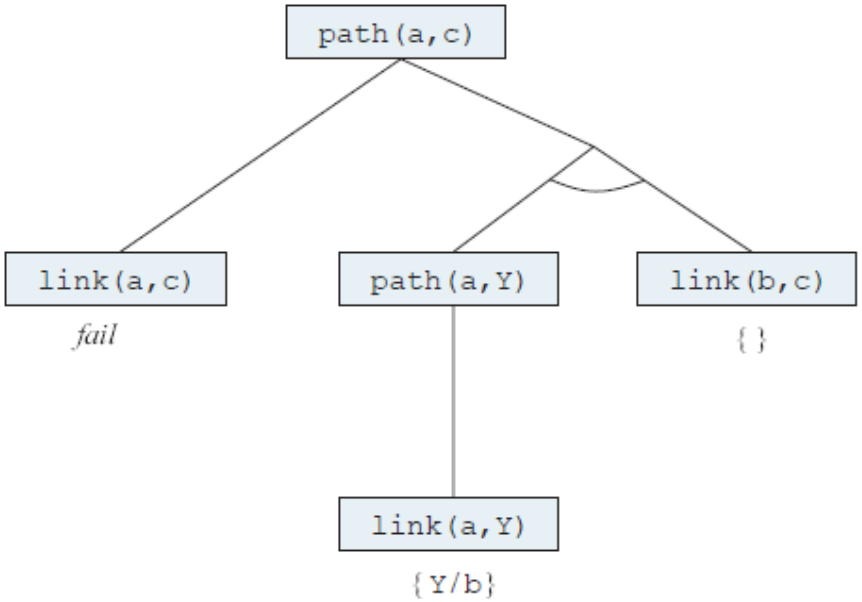
Infinite Loop

Finds if a path exists between two nodes in a directed graph.

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



Query `path(a, c)`



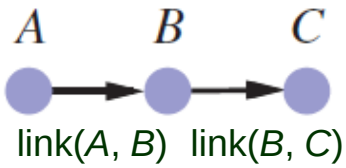
Infinite Loop

Finds if a path exists between two nodes in a directed graph.

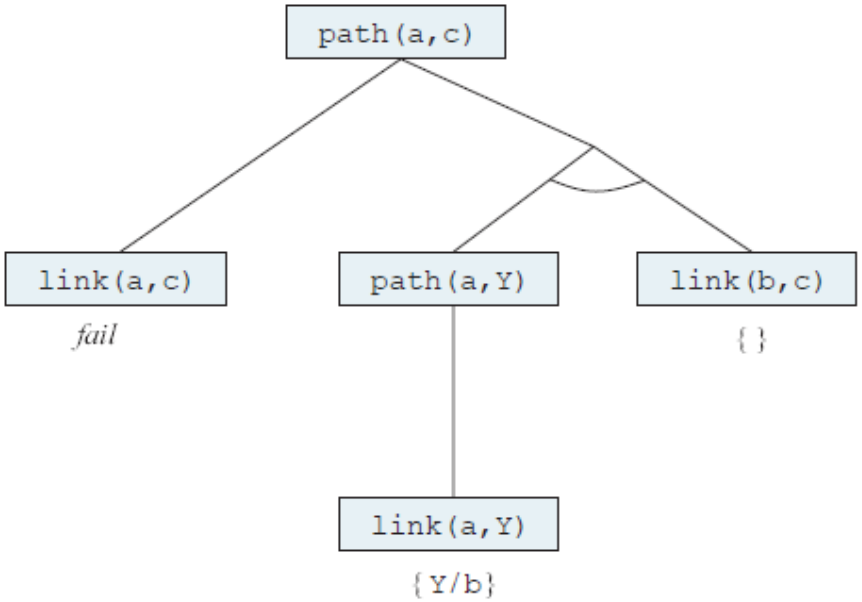
```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

switch order

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



Query `path(a, c)`



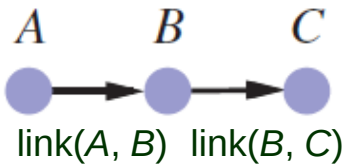
Infinite Loop

Finds if a path exists between two nodes in a directed graph.

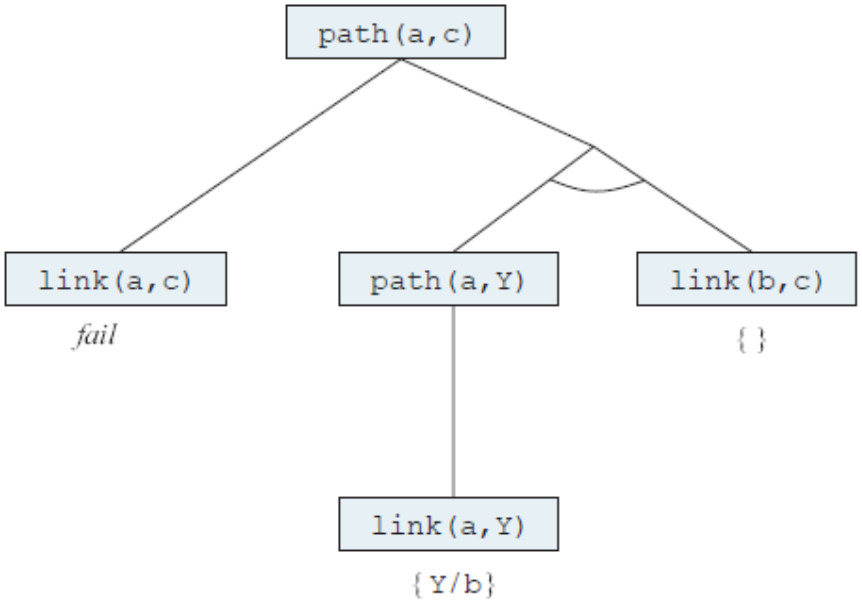
```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

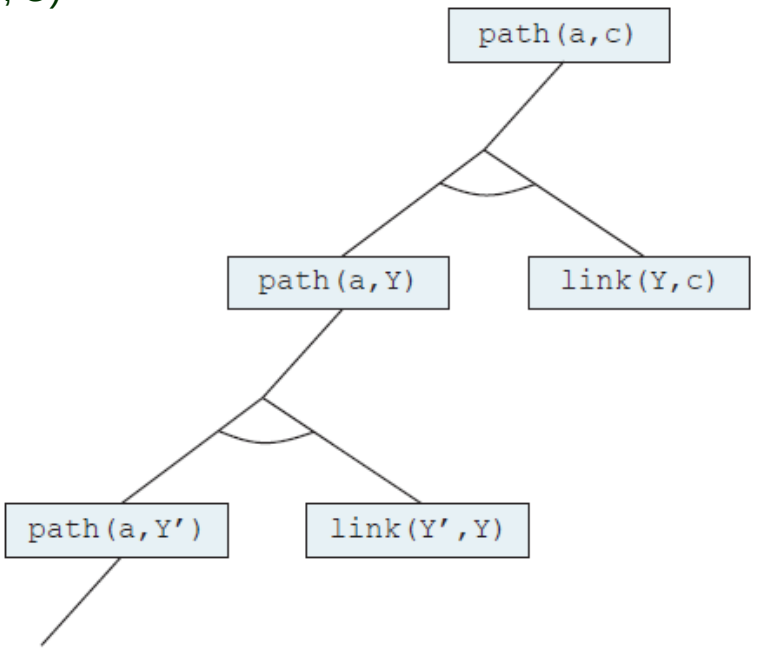
switch order

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



Query `path(a, c)`





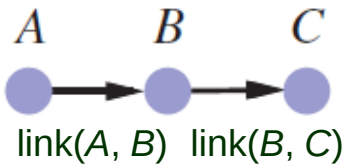
Infinite Loop

Finds if a path exists between two nodes in a directed graph.

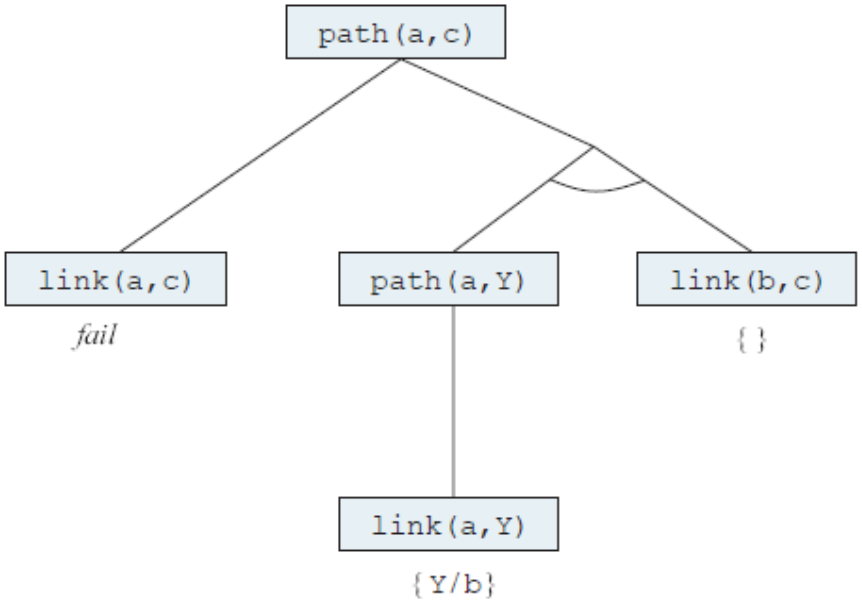
```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

switch order

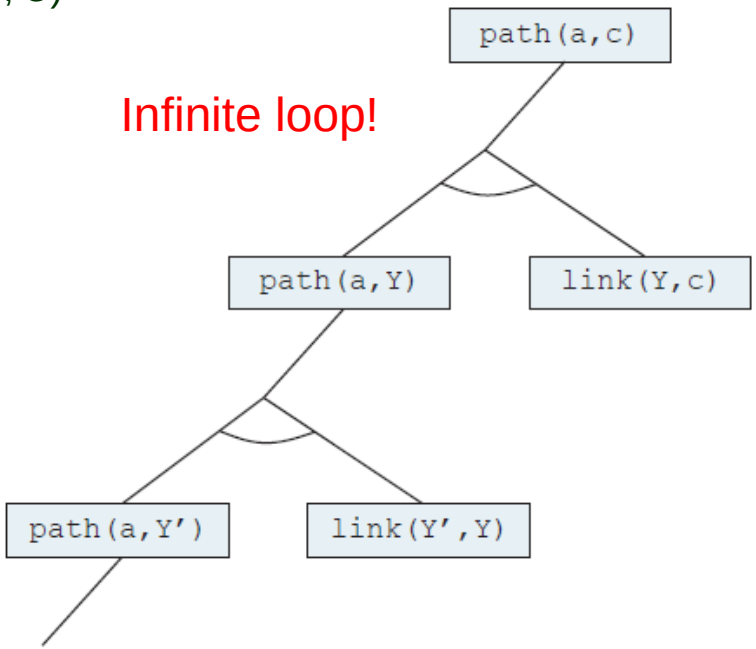
```
path(X,Z) :- path(X,Y), link(Y,Z).
path(X,Z) :- link(X, Z).
```



Query `path(a, c)`

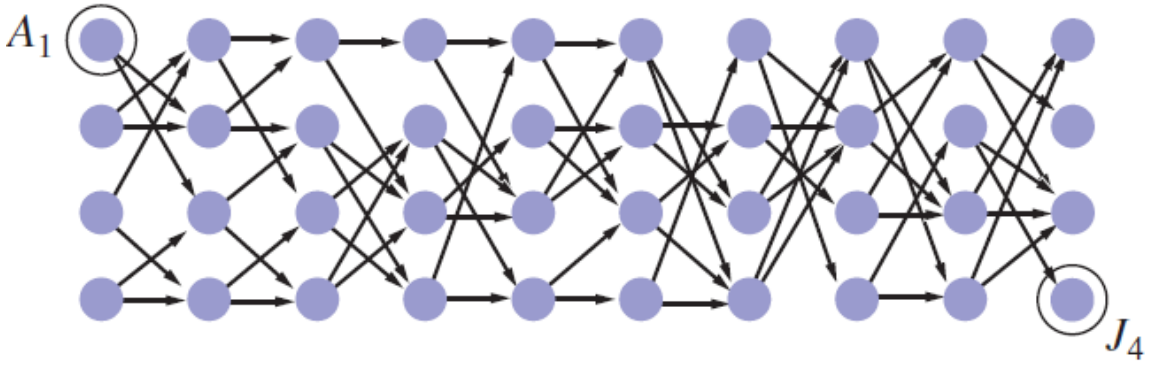


Infinite loop!



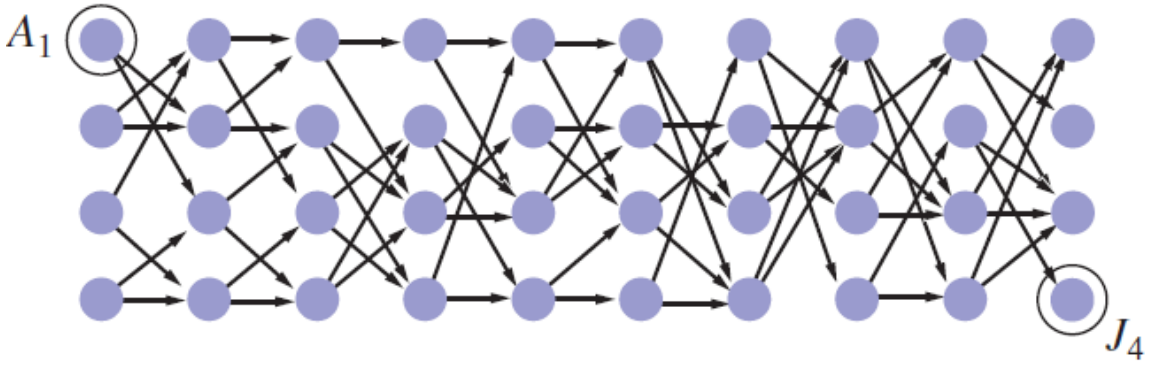
Redundant Inference

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



Redundant Inference

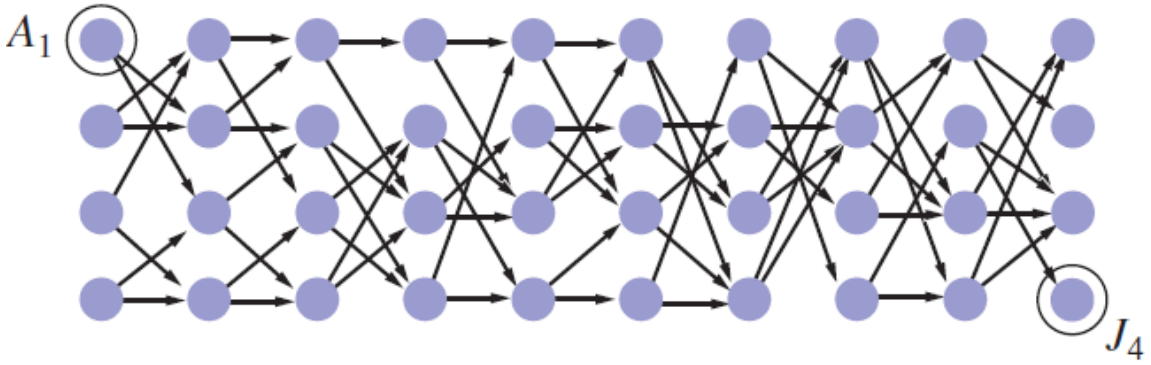
```
path(X,Z) :- link(X, Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```



Query `path(A1, J4)`

Redundant Inference

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

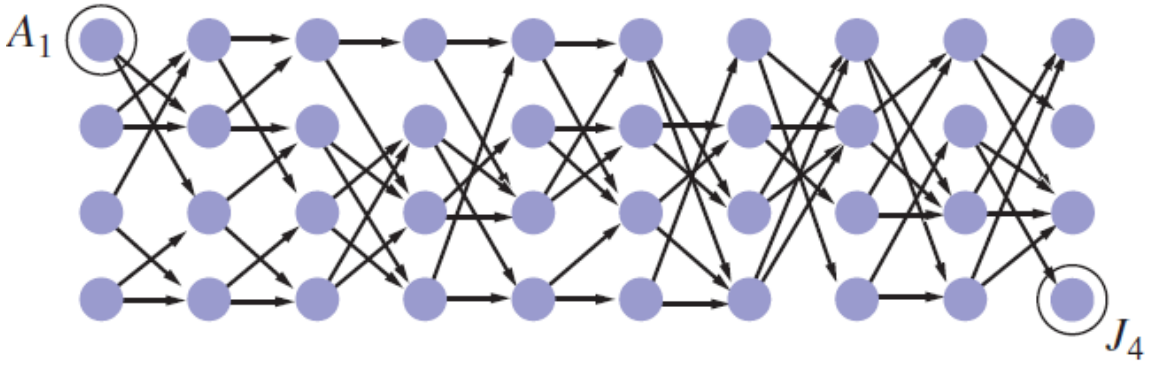


Query `path(A1, J4)`

- Prolog performs 877 inferences (most of which involve nodes from which the goal is unreachable).

Redundant Inference

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



Query `path(A1, J4)`

- ♠ Prolog performs 877 inferences (most of which involve nodes from which the goal is unreachable).
- ♦ Forward chaining performs only 62 inferences.

II. Resolution Inference Rule

$$\frac{l_1 \vee \cdots \vee l_i \vee \cdots \vee l_k, \qquad m_1 \vee \cdots \vee m_j \vee \cdots \vee m_k}{\text{SUBST}(\theta, l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where θ = UNIFY (l_i, m_j) makes l_i and m_j two complementary literals.

II. Resolution Inference Rule

$$\frac{l_1 \vee \cdots \vee l_i \vee \cdots \vee l_k, \qquad m_1 \vee \cdots \vee m_j \vee \cdots \vee m_k}{\text{SUBST}(\theta, l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where θ = UNIFY (l_i, m_j) makes l_i and m_j two complementary literals.

$Animal(F(x)) \vee Loves(G(x), x)$ $\qquad$ $(\neg Loves(u, v) \vee \neg Kills(u, v))$

II. Resolution Inference Rule

$$\frac{l_1 \vee \cdots \vee l_i \vee \cdots \vee l_k, \qquad m_1 \vee \cdots \vee m_j \vee \cdots \vee m_k}{\text{SUBST}(\theta, l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where θ = UNIFY (l_i, m_j) makes l_i and m_j two complementary literals.

$$\textit{Animal}(F(x)) \vee \underbrace{\textit{Loves}(G(x), x) \qquad (\neg \textit{Loves}(u, v)}_{\text{unifier: } \theta = \{u/G(x), v/x\}} \vee \neg \textit{Kills}(u, v))$$

II. Resolution Inference Rule

$$\frac{l_1 \vee \cdots \vee l_i \vee \cdots \vee l_k, \qquad m_1 \vee \cdots \vee m_j \vee \cdots \vee m_k}{\text{SUBST}(\theta, l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where θ = UNIFY (l_i, m_j) makes l_i and m_j two complementary literals.

$$\frac{\textit{Animal}(F(x)) \vee \underbrace{\textit{Loves}(G(x), x) \qquad (\neg \textit{Loves}(u, v)}_{\text{unifier: } \theta = \{u/G(x), v/x\}} \vee \neg \textit{Kills}(u, v))}{\text{resolvent:} \quad \textit{Animal}(F(x)) \vee \neg \textit{Kills}(G(x), x))}$$

II. Resolution Inference Rule

$$\frac{l_1 \vee \cdots \vee l_i \vee \cdots \vee l_k, \qquad m_1 \vee \cdots \vee m_j \vee \cdots \vee m_k}{\text{SUBST}(\theta, l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where θ = UNIFY (l_i, m_j) makes l_i and m_j two complementary literals.

$$\frac{\textit{Animal}(F(x)) \vee \underbrace{\textit{Loves}(G(x), x) \qquad (\neg \textit{Loves}(u, v)}_{\text{unifier: } \theta = \{u/G(x), v/x\}} \vee \neg \textit{Kills}(u, v))}{\textit{Animal}(F(x)) \vee \neg \textit{Kills}(G(x), x)}$$

resolvent: $\textit{Animal}(F(x)) \vee \neg \textit{Kills}(G(x), x)$

- Binary resolution as given above does not yield a complete inference procedure.
- *Full resolution* does. It resolves subsets of literals in each clause that are unifiable.

Example Proof 1

The crime example:

$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x)$

$\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono)$

$\neg Missile(x) \vee Weapon(x)$

$\neg Enemy(x, America) \vee Hostile(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$American(West)$

$Enemy(Nono, America)$

Example Proof 1

The crime example:

$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x)$

$\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono)$

$\neg Missile(x) \vee Weapon(x)$

$\neg Enemy(x, America) \vee Hostile(x)$

$Owns(Nono, M_1)$

$Missile(M_1)$

$American(West)$

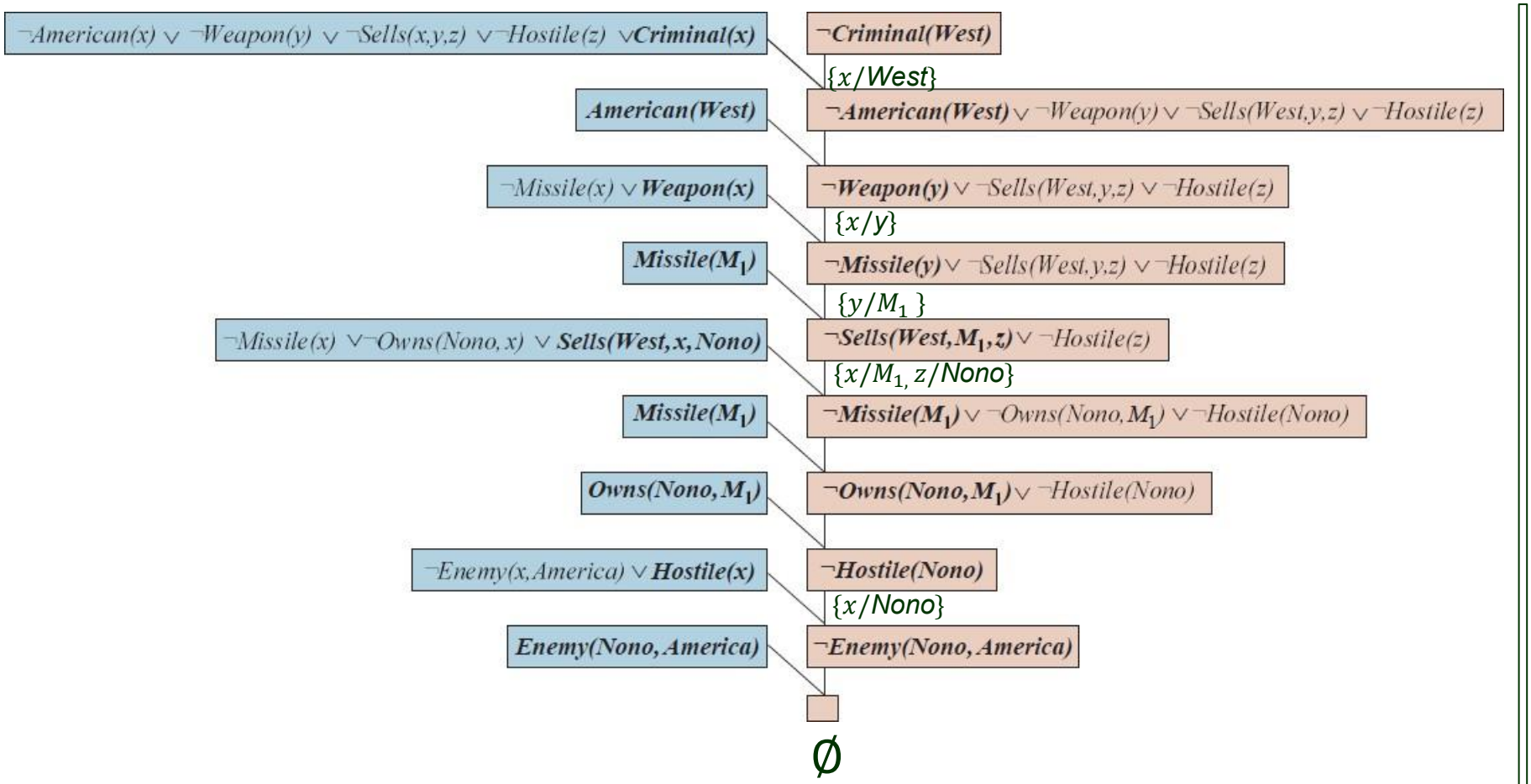
$Enemy(Nono, America)$

We prove *Criminal*(*West*) by adding

$\neg\ Criminal(West)$

and deriving the empty clause ∅.

Resolution Proof 1



- Like backward chaining on the main spine.
- Always resolve with a clause containing a positive literal that unifies with the leftmost literal of the "current" clause on the spine.

Example Proof 2 (with Skolemization)

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

Example Proof 2 (with Skolemization)

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

A. $\forall x \ \left(\forall y \ \mathit{Animal}(y) \Rightarrow \mathit{Loves}(x, y)\right) \Longrightarrow \exists y \ \mathit{Loves}(y, x)$

Example Proof 2 (with Skolemization)

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

A. $\forall x \ \left(\forall y \ Animal(y) \Rightarrow Loves(x, y)\right) \Longrightarrow \exists y \ Loves(y, x)$

B. $\forall x \ \left(\exists z \ Animal(z) \wedge Kills(x, z)\right) \Rightarrow (\forall y \ \neg Loves(y, x))$

Example Proof 2 (with Skolemization)

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

A. $\forall x \ \left(\forall y \ Animal(y) \Rightarrow Loves(x, y)\right) \Longrightarrow \exists y \ Loves(y, x)$

B. $\forall x \ \left(\exists z \ Animal(z) \wedge Kills(x, z)\right) \Rightarrow (\forall y \ \neg Loves(y, x))$

C. $\forall x \ \ Animal(x) \Rightarrow Loves(Jack, x)$

Example Proof 2 (with Skolemization)

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

A. $\forall x \ \left(\forall y \ Animal(y) \Rightarrow Loves(x, y)\right) \Longrightarrow \exists y \ Loves(y, x)$

B. $\forall x \ \left(\exists z \ Animal(z) \wedge Kills(x, z)\right) \Rightarrow (\forall y \ \neg Loves(y, x))$

C. $\forall x \ \ Animal(x) \Rightarrow Loves(Jack, x)$

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

Example Proof 2 (with Skolemization)

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

A. $\forall x \ \left(\forall y \ Animal(y) \Rightarrow Loves(x, y)\right) \Longrightarrow \exists y \ Loves(y, x)$

B. $\forall x \ \left(\exists z \ Animal(z) \wedge Kills(x, z)\right) \Rightarrow (\forall y \ \neg Loves(y, x))$

C. $\forall x \ \ Animal(x) \Rightarrow Loves(Jack, x)$

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

Background knowledge

E. $Cat(Tuna)$

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

Example Proof 2 (with Skolemization)

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

A. $\forall x \ \left(\forall y \ Animal(y) \Rightarrow Loves(x, y)\right) \Longrightarrow \exists y \ Loves(y, x)$

B. $\forall x \ \left(\exists z \ Animal(z) \wedge Kills(x, z)\right) \Rightarrow (\forall y \ \neg Loves(y, x))$

C. $\forall x \ \ Animal(x) \Rightarrow Loves(Jack, x)$

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

Background knowledge

E. $Cat(Tuna)$

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

Negated goal

$\neg$ G. $\neg Kills(Curiosity, Tuna)$

Converting to CNF

A. $\forall x \quad (\forall y \ \mathit{Animal}(y) \Rightarrow \mathit{Loves}(x, y)) \Rightarrow (\exists y \ \mathit{Loves}(x, y))$

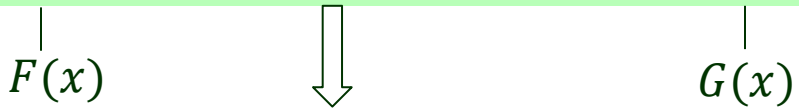
Converting to CNF

A. $\forall x \quad (\forall y \ \mathit{Animal}(y) \Rightarrow \mathit{Loves}(x, y)) \Rightarrow (\exists y \ \mathit{Loves}(x, y))$

$F(x)$ $\qquad$ $G(x)$

Converting to CNF

A. $\forall x \quad (\forall y \; Animal(y) \Rightarrow Loves(x, y)) \Rightarrow (\exists y \; Loves(x, y))$



A1. $Animal(F(x)) \vee Loves(G(x), x)$

A2. $\neg Loves(x, F(x)) \vee Loves(G(x), x)$

Converting to CNF

A. $\forall x \quad (\forall y \ Animal(y) \Rightarrow Loves(x, y)) \Rightarrow (\exists y \ Loves(x, y))$

$F(x)$ $G(x)$

A1. $Animal(F(x)) \vee Loves(G(x), x)$

A2. $\neg Loves(x, F(x)) \vee Loves(G(x), x)$

B. $\neg Animal(z) \vee \neg Kills(x, z) \vee \neg Loves(y, x)$

C. $\neg Animal(x) \vee Loves(Jack, x)$

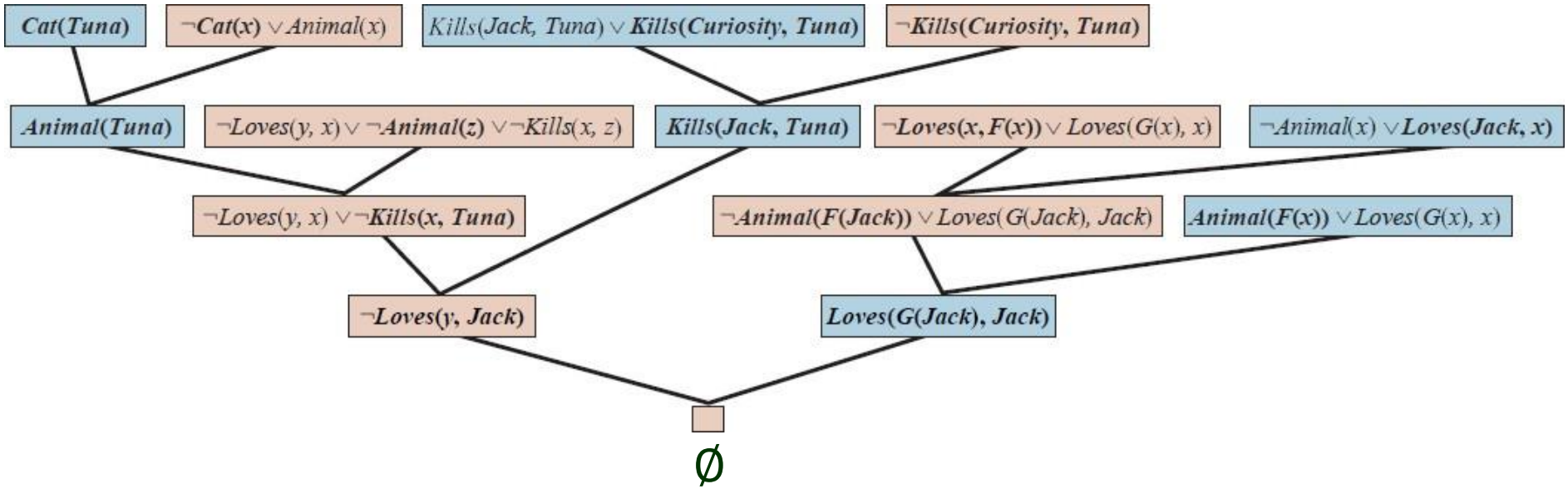
D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

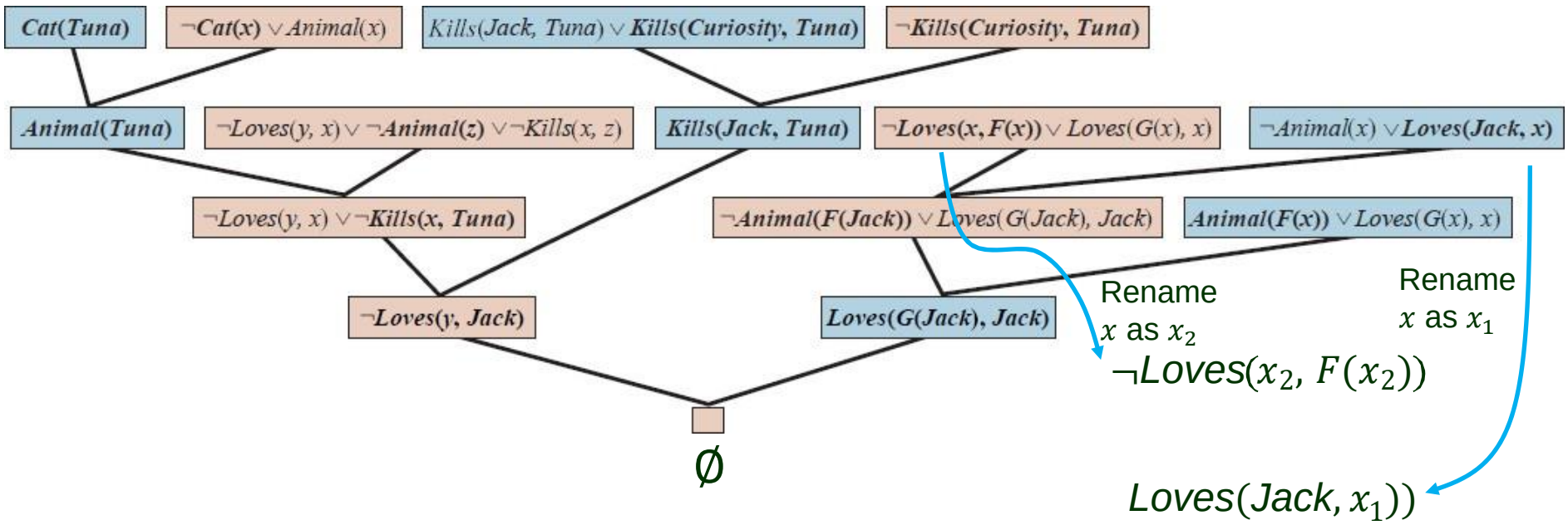
F. $\neg Cat(x) \vee Animal(x)$

¬ G. $\neg Kills(Curiosity, Tuna)$

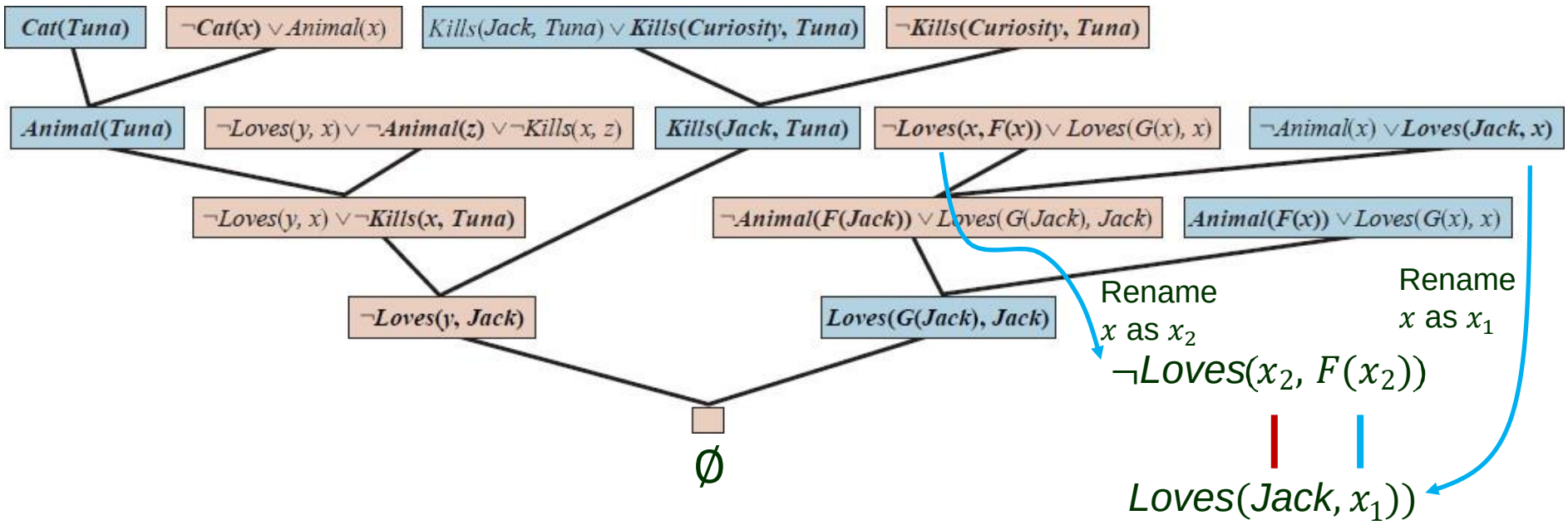
Resolution Proof 2



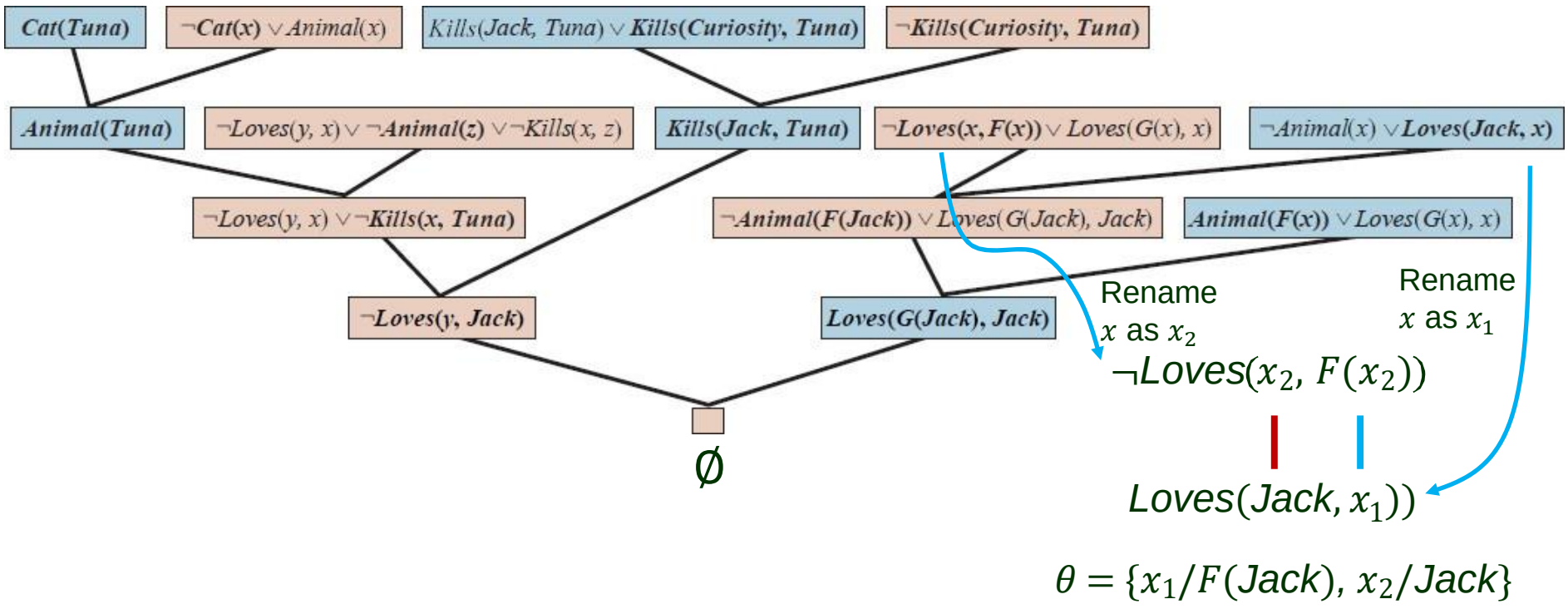
Resolution Proof 2



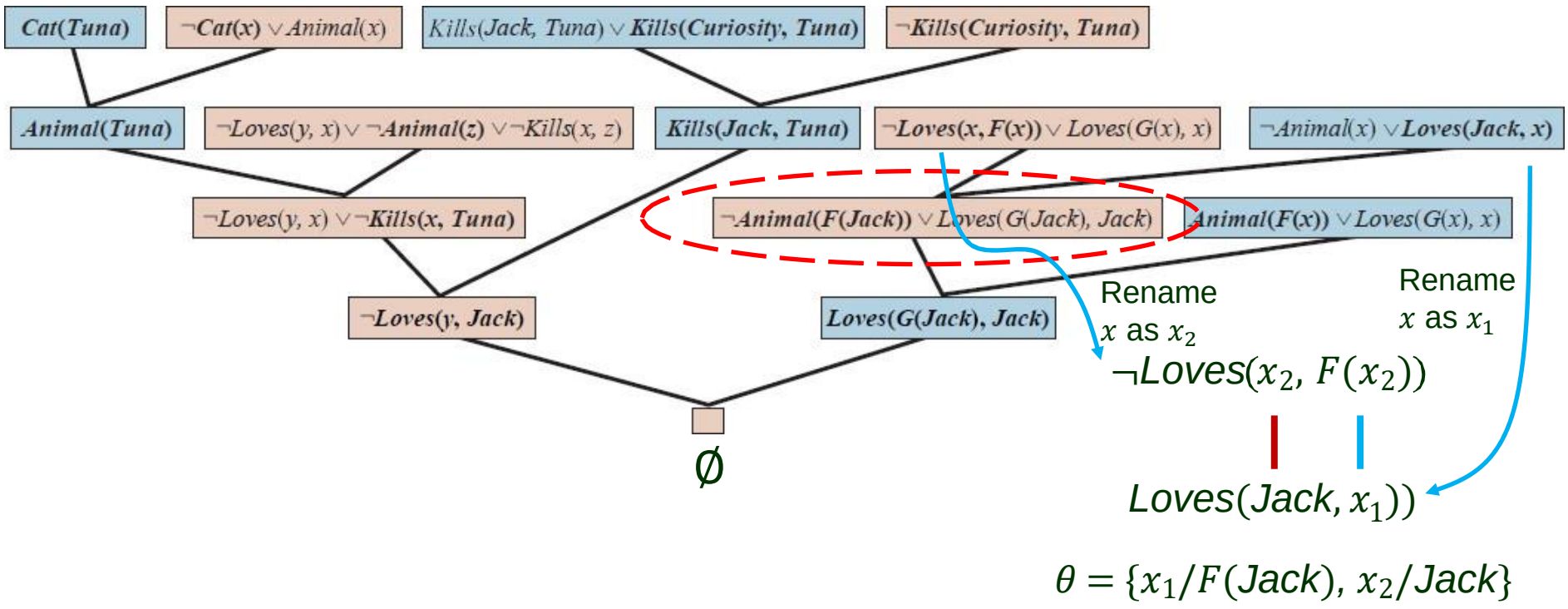
Resolution Proof 2



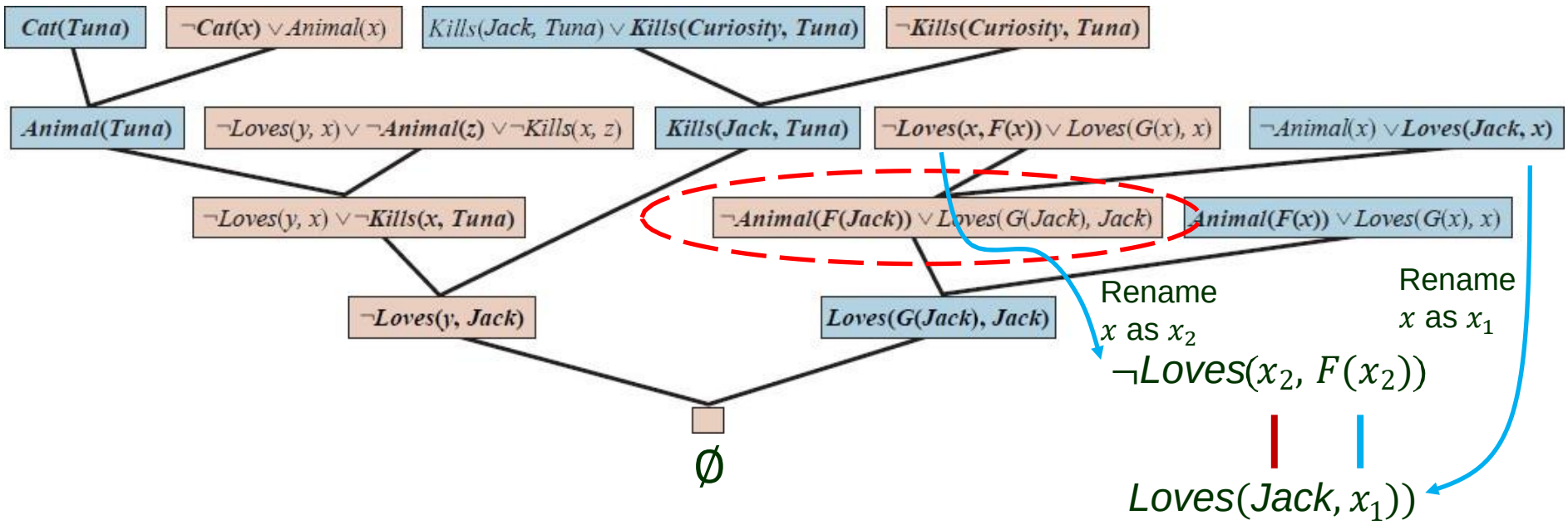
Resolution Proof 2



Resolution Proof 2



Resolution Proof 2



Paraphrased in English:

$\theta = \{x_1/F(Jack), x_2/Jack\}$

Suppose Curiosity did not kill Tuna. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore, Curiosity killed the cat.

Completeness of Resolution

Theorem If a set S of sentences is unsatisfiable, then resolution will always be able to derive a contradiction.

Not all logical consequences of S can be generated using resolution.

A sentence entailed by S can always be established using resolution.

Completeness of Resolution

Theorem If a set S of sentences is unsatisfiable, then resolution will always be able to derive a contradiction.

Not all logical consequences of S can be generated using resolution.

A sentence entailed by S can always be established using resolution.

We can use resolution to find all answers to a question $Q(x)$ by proving that $KB \wedge \neg Q(x)$ is unsatisfiable.