First-Order Logic

Introduction to Artificial Intelligence

Steven Bethard

Department of Computer Science
University of Colorado

CSCI 3202
Outline

1. First Order Logic
   - Motivation
   - Core Components
   - Quantifiers

2. Example Translations
   - Animal Kingdom
   - Wumpus World
   - Harry Potter

3. First-Order Logic Inference
   - Propositionalization
   - Unification

4. Inference Algorithms
   - Forward Chaining
   - Backward Chaining
   - Prolog
   - Resolution
Outline

1. First Order Logic
   - Motivation
   - Core Components
   - Quantifiers

2. Example Translations
   - Animal Kingdom
   - Wumpus World
   - Harry Potter

3. First-Order Logic Inference
   - Propositionalization
   - Unification

4. Inference Algorithms
   - Forward Chaining
   - Backward Chaining
   - Prolog
   - Resolution
What’s Wrong with Propositional Logic?

Translate:

All squares adjacent to pits are breezy
What’s Wrong with Propositional Logic?

Translate:

All squares adjacent to pits are breezy

Problem: Propositional Logic

\[ B_{1,1} \iff (P_{1,2} \lor P_{2,1}) \]
\[ B_{1,2} \iff (P_{1,1} \lor P_{2,2} \lor P_{1,3}) \]
\[ B_{2,1} \iff (P_{1,1} \lor P_{2,2} \lor P_{2,1}) \]

...
What’s Wrong with Propositional Logic?

Translate:

*All squares adjacent to pits are breezy*

Problem: Propositional Logic

\[
B_{1,1} \iff (P_{1,2} \lor P_{2,1}) \\
B_{1,2} \iff (P_{1,1} \lor P_{2,2} \lor P_{1,3}) \\
B_{2,1} \iff (P_{1,1} \lor P_{2,2} \lor P_{2,1})
\]

... 

Solution: First Order Logic (Preview)

\[
\forall s \ Breezy(s) \iff \exists r \ Adjacent(r, s) \land Pit(r)
\]
Propositional Logic

- World consists of facts
- All facts are either true or false
Core Ideas of First Order Logic

<table>
<thead>
<tr>
<th>Propositional Logic</th>
<th>First Order Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>World consists of <em>facts</em></td>
<td>World consists of <em>objects</em> and <em>relations</em></td>
</tr>
<tr>
<td>All facts are either true or false</td>
<td>Statements that a relation $R$ holds between objects $X_1, \ldots, X_n$ are either true or false</td>
</tr>
</tbody>
</table>
Core Ideas of First Order Logic

Propositional Logic
- World consists of facts
- All facts are either true or false

First Order Logic
- World consists of objects and relations
- Statements that a relation $R$ holds between objects $X_1, \ldots, X_n$ are either true or false
  
  **Objects**  people, numbers, houses, colors, years...
  **Relations** blonde, round, prime, multi-storied...
  brother-of, comes-between, has-color, after...
Constants

Key Idea

**Constants** represent named objects in the world

Examples

*RichardTheLionheart*

*RonaldMcDonald*

*Blue*

*42*

*12pm*
### Key Idea

**Functions** relate object(s) to *exactly one* other object

### Examples

- `LeftLegOf(x)`
- `LengthOf(x)`
- `SquareRoot(x)`, i.e. $\sqrt{x}$
- `Sum(x, y)`, i.e. $x + y$
- `Intersection(x, y)`, i.e. $x \cap y$
Predicates

Key Idea

**Predicates** describe relations between objects (or a property of a single object)

Examples

- $\text{Person}(x)$
- $\text{Female}(x)$
- $\text{BrotherOf}(x, y)$
- $\text{Positive}(x)$, i.e. $x > 0$
- $\text{MemberOf}(x, y)$, i.e. $x \in y$
- $\text{SubsetOf}(x, y)$, i.e. $x \subseteq y$
Connectives

Connectives in First-Order Logic

- Used to construct more complex sentences
- Semantics match those of Propositional Logic

Examples

\( \neg \text{Brother}(\text{LeftLeg}(\text{Richard}), \text{John}) \)

\( \text{Positive}(42) \land \text{LessThan}(42, 100) \)

\( \text{Male}(\text{Pat}) \lor \text{Female}(\text{Pat}) \)

\( \neg \text{LaysEggs}(\text{Whale}) \Rightarrow \neg \text{Bird}(\text{Whale}) \)
Truth in First Order Logic

Key Ideas
- Relations are (possibly infinite) sets of tuples
- $R(Term_1, \ldots, Term_n)$ is true iff $\langle Term_1, \ldots, Term_n \rangle \in R$

Example
Given the relation:
$SquareRoot = \{ \langle 1, 1 \rangle, \langle 4, 2 \rangle, \langle 9, 3 \rangle, \ldots \}$
Truth in First Order Logic

Key Ideas
- Relations are (possibly infinite) sets of tuples
- \( R(Term_1, \ldots, Term_n) \) is true iff \( \langle Term_1, \ldots, Term_n \rangle \in R \)

Example
Given the relation:

\[ \text{SquareRoot} = \{ \langle 1, 1 \rangle, \langle 4, 2 \rangle, \langle 9, 3 \rangle, \ldots \} \]

Then \( \text{SquareRoot}(4, 2) \) is true because \( \langle 4, 2 \rangle \in \text{SquareRoot} \).
Truth in First Order Logic

Key Ideas
- Relations are (possibly infinite) sets of tuples
- \( R(\text{Term}_1, \ldots, \text{Term}_n) \) is true iff \( \langle \text{Term}_1, \ldots, \text{Term}_n \rangle \in R \)

Example
Given the relation:

\( \text{SquareRoot} = \{ \langle 1, 1 \rangle, \langle 4, 2 \rangle, \langle 9, 3 \rangle, \ldots \} \)

Then \( \text{SquareRoot}(4, 2) \) is true because
\( \langle 4, 2 \rangle \in \text{SquareRoot} \)
Truth in First Order Logic

Key Ideas
- Relations are (possibly infinite) sets of tuples
- $R(Term_1, \ldots, Term_n)$ is true iff $\langle Term_1, \ldots, Term_n \rangle \in R$

Example
Given the relation:

$SquareRoot = \{ \langle 1, 1 \rangle, \langle 4, 2 \rangle, \langle 9, 3 \rangle, \ldots \}$

Then $SquareRoot(4, 2)$ is true because

$\langle 4, 2 \rangle \in SquareRoot$

And $SquareRoot(2, 2)$ is
Truth in First Order Logic

**Key Ideas**
- Relations are (possibly infinite) sets of tuples
- \( R(Term_1, \ldots, Term_n) \) is true iff \( \langle Term_1, \ldots, Term_n \rangle \in R \)

**Example**

Given the relation:

\[
\text{SquareRoot} = \{\langle 1, 1 \rangle, \langle 4, 2 \rangle, \langle 9, 3 \rangle, \ldots \}
\]

Then \( \text{SquareRoot}(4, 2) \) is true because

\( \langle 4, 2 \rangle \in \text{SquareRoot} \)

And \( \text{SquareRoot}(2, 2) \) is false because

\( \langle 2, 2 \rangle \notin \text{SquareRoot} \)
Predicating over Constants

If I know:
\[ \neg LaysEggs(Whale) \Rightarrow \neg Bird(Whale) \]

What can I say here?
\[ \neg LaysEggs(Steve) \Rightarrow ? \]
Quantifiers

Predicating over Constants

If I know:

\[ \neg \text{LaysEggs}(\text{Whale}) \implies \neg \text{Bird}(\text{Whale}) \]

What can I say here?

\[ \neg \text{LaysEggs}(\text{Steve}) \implies ? \]

Nothing!
Quantifiers

Predicating over Constants

If I know:
\[ \neg LaysEggs(Whale) \implies \neg Bird(Whale) \]

What can I say here?
\[ \neg LaysEggs(Steve) \implies ? \]

Nothing!

Quantifiers and Variables

Quantifiers allow statements about classes of objects, e.g.
\[ \forall x \neg LaysEggs(x) \implies \neg Bird(x) \]
Universal Quantification (\(\forall\))

**Definition**

\(\forall x \ P\) is true in model \(m\) iff:

- \(P\) is true when we bind \(x\) to each of the objects in \(m\)
Universal Quantification ($\forall$)

Definition

$\forall x \ P$ is true in model $m$ iff:

$P$ is true when we bind $x$ to each of the objects in $m$

Example

So $\forall x \ Bird(x) \Rightarrow LaysEggs(x)$ is true because:

$Bird(\text{Swallow}) \Rightarrow LaysEggs(\text{Swallow})$ is true

$Bird(\text{Emu}) \Rightarrow LaysEggs(\text{Emu})$ is true

$Bird(\text{Badger}) \Rightarrow LaysEggs(\text{Badger})$ is true

...
Universal Quantification (∀)

**Definition**

∀x \( P \) is true in model \( m \) iff:

\( P \) is true when we bind \( x \) to each of the objects in \( m \)

**Example**

So \( ∀x \text{ Bird}(x) \Rightarrow \text{LaysEggs}(x) \) is true because:

\( \text{Bird}(\text{Swallow}) \Rightarrow \text{LaysEggs}(\text{Swallow}) \) is true

\( \text{Bird}(\text{Emu}) \Rightarrow \text{LaysEggs}(\text{Emu}) \) is true

\( \text{Bird}(\text{Badger}) \Rightarrow \text{LaysEggs}(\text{Badger}) \) is true

\[ \ldots \]

But \( ∀x \text{ LaysEggs}(x) \Rightarrow \text{Bird}(x) \) is false because:
Universal Quantification (\(\forall\))

**Definition**

\(\forall x\ P\) is true in model \(m\) iff:

\(P\) is true when we bind \(x\) to each of the objects in \(m\)

**Example**

So \(\forall x\ Bird(x) \Rightarrow LaysEggs(x)\) is true because:

- \(Bird(Swallow) \Rightarrow LaysEggs(Swallow)\) is true
- \(Bird(Emu) \Rightarrow LaysEggs(Emu)\) is true
- \(Bird(Badger) \Rightarrow LaysEggs(Badger)\) is true

... 

But \(\forall x\ LaysEggs(x) \Rightarrow Bird(x)\) is false because:

- \(LaysEggs(Platypus) \Rightarrow Bird(Platypus)\) is false
Common Mistake

What does this mean?

$$\forall x \ Bird(x) \land LaysEggs(x)$$
Universal Quantification in Translation

Common Mistake
What does this mean?

\[ \forall x \ Bird(x) \land LaysEggs(x) \]

Answer:

*Everything is a bird and everything lays eggs*

Intended statement:

\[ \forall x \ Bird(x) \Rightarrow LaysEggs(x) \]
Universal Quantification in Translation

Common Mistake

What does this mean?

\[ \forall x \; Bird(x) \land LaysEggs(x) \]

Answer:

*Everything is a bird and everything lays eggs*

Intended statement:

\[ \forall x \; Bird(x) \Rightarrow LaysEggs(x) \]

Rule of Thumb

Use implication (\( \Rightarrow \)) with universal quantification (\( \forall \))
Existential Quantification

**Definition**

\[ \exists x \ P \text{ is true in model } m \text{ iff: } \]

\[ P \text{ is true when we bind } x \text{ to any of the objects in } m \]
Existential Quantification

Definition

$\exists x \ P$ is true in model $m$ iff:

$P$ is true when we bind $x$ to any of the objects in $m$

Example

So $\exists x \ Mammal(x) \land LaysEggs(x)$ is true because:
Existential Quantification

**Definition**

\[ \exists x \ P \text{ is true in model } m \text{ iff:} \]

\[ P \text{ is true when we bind } x \text{ to any of the objects in } m \]

**Example**

So \[ \exists x \ Mammal(x) \land LaysEggs(x) \] is true because:

\[ Mammal(\text{Platypus}) \land LaysEggs(\text{Platypus}) \] is true
**Existential Quantification**

**Definition**

\( \exists x \ P \) is true in model \( m \) iff:

\( P \) is true when we bind \( x \) to any of the objects in \( m \)

**Example**

So: \( \exists x \ Mammal(x) \land LaysEggs(x) \) is true because:

\( Mammal(Platypus) \land LaysEggs(Platypus) \) is true

But: \( \exists x \ Mammal(x) \land \neg Animal(x) \) is false because:
Existential Quantification

**Definition**

\[ \exists x \ P \text{ is true in model } m \text{ iff:} \]

\[ P \text{ is true when we bind } x \text{ to any of the objects in } m \]

**Example**

So \( \exists x \ Mammal(x) \land LaysEggs(x) \) is true because:

\[ Mammal(Platypus) \land LaysEggs(Platypus) \] is true

But \( \exists x \ Mammal(x) \land \neg Animal(x) \) is false because:

\[ Mammal(Person) \land \neg Animal(Person) \] is false
\[ Mammal(Platypus) \land \neg Animal(Platypus) \] is false
\[ Mammal(Spam) \land \neg Animal(Spam) \] is false

...
Common Mistake
What does this mean?

$$\exists x \ Mammal(x) \Rightarrow LaysEggs(x)$$
Common Mistake

What does this mean?

\[ \exists x \ Mammal(x) \Rightarrow LaysEggs(x) \]

Answer:

There is something that is not a mammal or lays eggs

Intended statement:

\[ \exists x \ Mammal(x) \land LaysEggs(x) \]
**Existential Quantification in Translation**

**Common Mistake**

What does this mean?

\[ \exists x \ Mammal(x) \Rightarrow LaysEggs(x) \]

Answer:

There is something that is not a mammal or lays eggs

Intended statement:

\[ \exists x \ Mammal(x) \land LaysEggs(x) \]

**Rule of Thumb**

Use conjunction (\( \land \)) with existential quantification (\( \exists \))
Quantifier Properties

Nesting Quantifiers

Mixed quantifiers cannot be exchanged:

- $\forall x \exists y \text{Loves}(x, y)$ “everyone loves someone”
- $\exists x \forall y \text{Loves}(x, y)$ “one person is loved by everyone”
### Quantifier Properties

#### Nesting Quantifiers

Mixed quantifiers cannot be exchanged:

- $\forall x \exists y \text{Loves}(x, y)$ “everyone loves someone”
- $\exists x \forall y \text{Loves}(x, y)$ “one person is loved by everyone”

#### Relation between $\forall$ and $\exists$

Conversion is roughly like DeMorgan’s:

- $\forall x \text{Enjoys}(x, AI) \equiv \neg \exists x \neg \text{Enjoys}(x, AI)$
- $\exists x \text{Enjoys}(x, DB) \equiv \neg \forall x \neg \text{Enjoys}(x, DB)$
Equality Relations

Problem

What’s wrong with this definition?

\[ \forall x, y \ Sibling(x, y) \iff \exists p \ Parent(p, x) \land Parent(p, y) \]

Both \( x \) and \( y \) can be the same thing!

Solution: Equality Specify when two variables refer to the same objects:

\[ \forall x, y \ Sibling(x, y) \iff x \neq y \land \exists p \ Parent(p, x) \land Parent(p, y) \]
Equality Relations

Problem

What’s wrong with this definition?
\[ \forall x, y \ Sibling(x, y) \iff \exists p \ Parent(p, x) \land Parent(p, y) \]

Both \( x \) and \( y \) can be the same thing!
\[ \text{Sibling}(Steve, Steve) \]

Solution: Equality
Specify when two variables refer to the same objects:
\[ \forall x, y \ Sibling(x, y) \iff x \neq y \land \exists p \ Parent(p, x) \land Parent(p, y) \]
Problem

What’s wrong with this definition?

\[
\forall x, y \ Sibling(x, y) \iff \exists p \ Parent(p, x) \land Parent(p, y)
\]

Both \(x\) and \(y\) can be the same thing!

\(Sibling(Steve, Steve)\)

Solution: Equality

Specify when two variables refer to the same objects:

\[
\forall x, y \ Sibling(x, y) \iff x \neq y \land \exists p \ Parent(p, x) \land Parent(p, y)
\]
Outline

1. First Order Logic
   - Motivation
   - Core Components
   - Quantifiers

2. Example Translations
   - Animal Kingdom
   - Wumpus World
   - Harry Potter

3. First-Order Logic Inference
   - Propositionalization
   - Unification

4. Inference Algorithms
   - Forward Chaining
   - Backward Chaining
   - Prolog
   - Resolution
Animal Kingdom Translations

- All horses are mammals
Animal Kingdom Translations

- All horses are mammals
  \[ \forall x \text{ Horse}(x) \Rightarrow \text{Mammal}(x) \]
Animal Kingdom Translations

- All horses are mammals
  \[ \forall x \text{ Horse}(x) \Rightarrow \text{Mammal}(x) \]
- All birds have wings
Animal Kingdom Translations

- All horses are mammals
  \( \forall x \; \text{Horse}(x) \implies \text{Mammal}(x) \)
- All birds have wings
  \( \forall x \; \text{Bird}(x) \implies \text{HasWings}(x) \)
All horses are mammals
\[ \forall x \text{ Horse}(x) \Rightarrow \text{Mammal}(x) \]

All birds have wings
\[ \forall x \text{ Bird}(x) \Rightarrow \text{HasWings}(x) \]

Some mammals lay eggs
\[ \exists x \text{ Mammal}(x) \land \text{LaysEggs}(x) \]
All horses are mammals
\( \forall x \ Horse(x) \Rightarrow Mammal(x) \)

All birds have wings
\( \forall x \ Bird(x) \Rightarrow HasWings(x) \)

Some mammals lay eggs
\( \exists x \ Mammal(x) \land LaysEggs(x) \)
Animal Kingdom Translations

- All horses are mammals
  \[ \forall x \: \text{Horse}(x) \Rightarrow \text{Mammal}(x) \]
- All birds have wings
  \[ \forall x \: \text{Bird}(x) \Rightarrow \text{HasWings}(x) \]
- Some mammals lay eggs
  \[ \exists x \: \text{Mammal}(x) \land \text{LaysEggs}(x) \]
- Some birds don’t fly
Animal Kingdom Translations

- All horses are mammals
  \[ \forall x \ Horse(x) \Rightarrow Mammal(x) \]
- All birds have wings
  \[ \forall x \ Bird(x) \Rightarrow HasWings(x) \]
- Some mammals lay eggs
  \[ \exists x \ Mammal(x) \land LaysEggs(x) \]
- Some birds don’t fly
  \[ \exists x \ Bird(x) \land \neg Flies(x) \]
Animal Kingdom Translations

- All horses are mammals
  \( \forall x \ Horse(x) \Rightarrow Mammal(x) \)

- All birds have wings
  \( \forall x \ Bird(x) \Rightarrow HasWings(x) \)

- Some mammals lay eggs
  \( \exists x \ Mammal(x) \land LaysEggs(x) \)

- Some birds don’t fly
  \( \exists x \ Bird(x) \land \neg Flies(x) \)

- Animals that fly have wings
Animal Kingdom Translations

- All horses are mammals
  \[ \forall x \text{ Horse}(x) \Rightarrow \text{Mammal}(x) \]

- All birds have wings
  \[ \forall x \text{ Bird}(x) \Rightarrow \text{HasWings}(x) \]

- Some mammals lay eggs
  \[ \exists x \text{ Mammal}(x) \land \text{LaysEggs}(x) \]

- Some birds don’t fly
  \[ \exists x \text{ Bird}(x) \land \neg \text{Flies}(x) \]

- Animals that fly have wings
  \[ \forall x \text{ Animal}(x) \land \text{Flies}(x) \Rightarrow \text{HasWings}(x) \]
Animal Kingdom Translations

- All horses are mammals
  \( \forall x \, \text{Horse}(x) \Rightarrow \text{Mammal}(x) \)
- All birds have wings
  \( \forall x \, \text{Bird}(x) \Rightarrow \text{HasWings}(x) \)
- Some mammals lay eggs
  \( \exists x \, \text{Mammal}(x) \land \text{LaysEggs}(x) \)
- Some birds don’t fly
  \( \exists x \, \text{Bird}(x) \land \neg \text{Flies}(x) \)
- Animals that fly have wings
  \( \forall x \, \text{Animal}(x) \land \text{Flies}(x) \Rightarrow \text{HasWings}(x) \)
- Not all swimming animals have fins
Animal Kingdom Translations

- All horses are mammals
  \( \forall x \ Horse(x) \Rightarrow Mammal(x) \)

- All birds have wings
  \( \forall x \ Bird(x) \Rightarrow HasWings(x) \)

- Some mammals lay eggs
  \( \exists x \ Mammal(x) \land LaysEggs(x) \)

- Some birds don’t fly
  \( \exists x \ Bird(x) \land \neg Flies(x) \)

- Animals that fly have wings
  \( \forall x \ Animal(x) \land Flies(x) \Rightarrow HasWings(x) \)

- Not all swimming animals have fins
  \( \neg \forall x \ Animal(x) \land Swim(x) \Rightarrow HasFins(x) \)
Animal Kingdom Translations

- All horses are mammals
  \[ \forall x \ Horse(x) \Rightarrow Mammal(x) \]

- All birds have wings
  \[ \forall x \ Bird(x) \Rightarrow HasWings(x) \]

- Some mammals lay eggs
  \[ \exists x \ Mammal(x) \land LaysEggs(x) \]

- Some birds don’t fly
  \[ \exists x \ Bird(x) \land \neg Flies(x) \]

- Animals that fly have wings
  \[ \forall x \ Animal(x) \land Flies(x) \Rightarrow HasWings(x) \]

- Not all swimming animals have fins
  \[ \neg \forall x \ Animal(x) \land Swim(x) \Rightarrow HasFins(x) \]
  \[ \exists x \ Animal(x) \land Swim(x) \land \neg HasFins(x) \]
An animal gives birth to animals of the same species
Animal Kingdom Translations

- An animal gives birth to animals of the same species
  \[ \forall x, y \text{ Animal}(x) \land \text{GivesBirth}(x, y) \Rightarrow \text{Animal}(y) \land \text{Species}(x) = \text{Species}(y) \land x \neq y \]
Animal Kingdom Translations

- An animal gives birth to animals of the same species
  \[ \forall x, y \; \text{Animal}(x) \land \text{GivesBirth}(x, y) \implies \text{Animal}(y) \land \text{Species}(x) = \text{Species}(y) \land x \neq y \]

- Bats have exactly two wings
An animal gives birth to animals of the same species
\[ \forall x, y \text{ Animal}(x) \land \text{GivesBirth}(x, y) \Rightarrow \text{Animal}(y) \land \text{Species}(x) = \text{Species}(y) \land x \neq y \]

Bats have exactly two wings
\[ \forall x \text{ Bat}(x) \Rightarrow \exists y, z \text{ HasWing}(x, y) \land \text{HasWing}(x, z) \land y \neq z \land \forall w \text{ HasWing}(x, w) \Rightarrow w = y \lor w = z \]
Wumpus World Translations

- There is a breeze in [3, 1]
Wumpus World Translations

- There is a breeze in [3, 1]
  \( Breezy([3, 1]) \)
Wumpus World Translations

- There is a breeze in [3, 1]
  \( Breezy([3, 1]) \)

- The Wumpus is lives in [2, 2]
There is a breeze in [3, 1]  
\[ \text{Breezy}([3, 1]) \]

The Wumpus is lives in [2, 2]  
\[ \text{Home}(\text{Wumpus}) = [2, 2] \]
Wumpus World Translations

- There is a breeze in [3, 1]
  \( Breezy([3, 1]) \)

- The Wumpus is lives in [2, 2]
  \( Home(Wumpus) = [2, 2] \)

- If you are in the Wumpus’s square, he eats you
There is a breeze in [3, 1] 
\[ Breezy([3, 1]) \]

The Wumpus is lives in [2, 2] 
\[ Home(Wumpus) = [2, 2] \]

If you are in the Wumpus’s square, he eats you 
\[ \forall t \ Location(Agent, Home(Wumpus), t) \Rightarrow HasEaten(Wumpus, Agent, t) \]
Wumpus World Translations

- There is a breeze in [3, 1]
  \( \text{Breezy}([3, 1]) \)

- The Wumpus is lives in [2, 2]
  \( \text{Home}(\text{Wumpus}) = [2, 2] \)

- If you are in the Wumpus’s square, he eats you
  \( \forall t \ \text{Location}(\text{Agent}, \text{Home}(\text{Wumpus}), t) \Rightarrow \text{HasEaten}(\text{Wumpus}, \text{Agent}, t) \)

- You should grab the gold when you are in its square
Wumpus World Translations

- There is a breeze in [3, 1]
  \[ Breezy([3, 1]) \]

- The Wumpus is lives in [2, 2]
  \[ Home(Wumpus) = [2, 2] \]

- If you are in the Wumpus’s square, he eats you
  \[ \forall t \ Location(Agent, Home(Wumpus), t) \Rightarrow HasEaten(Wumpus, Agent, t) \]

- You should grab the gold when you are in its square
  \[ \forall s, t \ HasGold(s) \land Location(Agent, s, t) \Rightarrow BestAction(Grab, Agent, t) \]
Wumpus World Translations

Diagnostic Rules

From effect, determine cause:

$$\forall y \text{ Breezy}(y) \Rightarrow (\exists x \text{ Pit}(x) \land \text{Adjacent}(x, y))$$
Diagnostic Rules
From effect, determine cause:
\[ \forall y \ Breezy(y) \Rightarrow (\exists x \ Pit(x) \land Adjacent(x, y)) \]

Causal Rules
From cause, determine effect:
\[ \forall x, y \ (Pit(x) \land Adjacent(x, y)) \Rightarrow Breezy(y) \]
Wumpus World Translations

Diagnostic Rules
From effect, determine cause:
\[ \forall y \ Breezy(y) \Rightarrow (\exists x \ Pit(x) \land Adjacent(x, y)) \]

Causal Rules
From cause, determine effect:
\[ \forall x, y \ (Pit(x) \land Adjacent(x, y)) \Rightarrow Breezy(y) \]

Definition Rules
Bidirectional:
\[ \forall y \ Breezy(y) \iff (\exists x \ Pit(x) \land Adjacent(x, y)) \]
Harry Potter’s 7 Potions Puzzle

Danger lies before you, while safety lies behind,  
Two of us will help you, whichever you would find.  
One among us seven will let you move ahead,  
Another will transport the drinker back instead.  
Two among our number hold only nettle wine,  
Three of us are killers, waiting hidden in line.  
Choose, unless you wish to stay here forevermore,  
To help you in your choice, we give you these clues four:  
First, however slyly the poison tries to hide  
You will always find some on nettle wine’s left side;  
Second, different are those who stand at either end,  
But if you would move forward, neither is your friend;  
Third, as you see clearly, all are different size,  
Neither dwarf nor giant holds death in their insides;  
Fourth, the second left and second on the right  
Are twins once you taste them, though different at first sight.
∀p \text{ Potions}(p) \iff \\
Permutation(p, [\text{Forward}, \text{Backward}, \text{Wine}, \text{Wine}, \text{Poison}, \text{Poison}, \text{Poison}]) \\
\text{PoisonIsLeftOfWine}(p) \land \\
\text{EndsAreDifferent}(p) \land \text{EndsAreNotForward}(p) \land \\
\text{SmallestIsNotPoison}(p) \land \text{LargestIsNotPoison}(p) \\
\text{SecondsAreTheSame}(p) \land \\
\ldots
One Harry Potter’s 7 Potions Solution

\[ \forall p \; \text{Potions}(p) \iff \text{Permutation}(p, [\text{Forward, Backward, Wine, Wine, Poison, Poison, Poison}]) \]

\[ \text{PoisonIsLeftOfWine}(p) \land \text{EndsAreDifferent}(p) \land \text{EndsAreNotForward}(p) \land \text{SmallestIsNotPoison}(p) \land \text{LargestIsNotPoison}(p) \land \text{SecondsAreTheSame}(p) \land \ldots \]

\[ \forall p \; \text{EndsAreDifferent}(p) \iff \exists p_1, p_2, \ldots, p_7 \; p = [p_1, p_2, p_3, p_4, p_5, p_6, p_7] \land p_1 \neq p_7 \land \ldots \]
One Harry Potter’s 7 Potions Solution

\[ \forall p \text{ Potions}(p) \Leftrightarrow \]
\[ \text{Permutation}(p, [\text{Forward, Backward, Wine, Wine, Poison, Poison, Poison}]) \]
\[ \text{PoisonIsLeftOfWine}(p) \land \]
\[ \text{EndsAreDifferent}(p) \land \text{EndsAreNotForward}(p) \land \]
\[ \text{SmallestIsNotPoison}(p) \land \text{LargestIsNotPoison}(p) \land \]
\[ \text{SecondsAreTheSame}(p) \land \]
\[ \ldots \]
\[ \forall p \text{ EndsAreDifferent}(p) \Leftrightarrow \]
\[ \exists p_1, p_2, \ldots, p_7 \ p = [p_1, p_2, p_3, p_4, p_5, p_6, p_7] \land p_1 \neq p_7 \]
\[ \ldots \]
\[ \forall p \text{ LargestIsNotPoison}(p) \Leftrightarrow \]
\[ \exists p_i \text{ Largest}(p, p_i) \land p_i \neq \text{Poison} \]
Prolog Demo
Outline

1. First Order Logic
   - Motivation
   - Core Components
   - Quantifiers

2. Example Translations
   - Animal Kingdom
   - Wumpus World
   - Harry Potter

3. First-Order Logic Inference
   - Propositionalization
   - Unification

4. Inference Algorithms
   - Forward Chaining
   - Backward Chaining
   - Prolog
   - Resolution
Universal Instantiation

Key Idea

If we know $\forall x \ P(x)$, then we can conclude:

- $P(Badger)$
- $P(Spam)$
- ...
Key Idea

If we know $\forall x \ P(x)$, then we can conclude:

- $P(\text{Badger})$
- $P(\text{Spam})$
- $\ldots$

Formal Rule

Given a ground term $g$:

$\forall \nu \ \alpha$

$\frac{}{\text{SUBST}(\{\nu/g\}, \alpha)}$
Key Idea

If we know $\exists x \ P(x)$, then we can just give a name to $x$:

$P(\text{ThingThatPlsTrueFor})$

This works as long as the name isn’t already in use
Existential Instantiation

Key Idea
If we know $\exists x \ P(x)$, then we can just give a name to $x$:

$P(\text{ThingThatPlsTrueFor})$

This works as long as the name isn’t already in use

Formal Rule
Given a constant $k$ that is not in the knowledge base:

$$\exists v \alpha$$

$$\text{SUBST}\left(\{v/k\}, \alpha\right)$$

The constant $k$ is called a Skolem constant
Reduction to Propositional Logic

First-Order Logic

\[ \exists x \ Mammal(x) \land LaysEggs(x) \]
\[ \forall x \ Mammal(x) \Rightarrow WarmBlooded(x) \]
\[ Mammal(\text{Platypus}) \]
\[ \neg WarmBlooded(\text{Crocodile}) \]
**Reduction to Propositional Logic**

**First-Order Logic**

\[ \exists x \ Mammal(x) \land LaysEggs(x) \]
\[ \forall x \ Mammal(x) \Rightarrow \text{WarmBlooded}(x) \]
\[ Mammal(\text{Platypus}) \]
\[ \neg \text{WarmBlooded}(\text{Crocodile}) \]

**Propositional Logic**

Mammal(EggLayer) \land LaysEggs(EggLayer)

Mammal(Platypus) \Rightarrow \text{WarmBlooded}(Platypus)

Mammal(Crocodile) \Rightarrow \text{WarmBlooded}(Crocodile)

Mammal(Platypus) \neg \text{WarmBlooded}(Crocodile)
First-Order Logic

∃x Mammal(x) ∧ LaysEggs(x)
∀x Mammal(x) ⇒ WarmBlooded(x)
Mammal(Platypus)
¬WarmBlooded(Crocodile)

Propositional Logic

Mammal(EggLayer) ∧ LaysEggs(EggLayer)
**Reduction to Propositional Logic**

**First-Order Logic**

\[
\begin{align*}
\exists x & \ Mammal(x) \land LaysEggs(x) \\
\forall x & \ Mammal(x) \Rightarrow WarmBlooded(x) \\
Mammal(Platypus) \\
\lnot WarmBlooded(Crocodile)
\end{align*}
\]

**Propositional Logic**

\[
\begin{align*}
Mammal(EggLayer) \land LaysEggs(EggLayer) \\
Mammal(Platypus) \Rightarrow WarmBlooded(Platypus) \\
Mammal(Crocodile) \Rightarrow WarmBlooded(Crocodile)
\end{align*}
\]
First-Order Logic

\[ \exists x \; \text{Mammal}(x) \land \text{LaysEggs}(x) \]
\[ \forall x \; \text{Mammal}(x) \Rightarrow \text{WarmBlooded}(x) \]
\[ \text{Mammal}(\text{Platypus}) \]
\[ \neg \text{WarmBlooded}(\text{Crocodile}) \]

Propositional Logic

\[ \text{Mammal}(\text{EggLayer}) \land \text{LaysEggs}(\text{EggLayer}) \]
\[ \text{Mammal}(\text{Platypus}) \Rightarrow \text{WarmBlooded}(\text{Platypus}) \]
\[ \text{Mammal}(\text{Crocodile}) \Rightarrow \text{WarmBlooded}(\text{Crocodile}) \]
\[ \text{Mammal}(\text{Platypus}) \]
Reduction to Propositional Logic

**First-Order Logic**

\[ \exists x \ Mammal(x) \land LaysEggs(x) \]
\[ \forall x \ Mammal(x) \Rightarrow WarmBlooded(x) \]

Mammal(Platypus)

\[ \neg WarmBlooded(Crocodile) \]

**Propositional Logic**

Mammal(EggLayer) \land LaysEggs(EggLayer)

Mammal(Platypus) \Rightarrow WarmBlooded(Platypus)

Mammal(Crocodile) \Rightarrow WarmBlooded(Crocodile)

Mammal(Platypus)

\[ \neg WarmBlooded(Crocodile) \]
Simple First-Order Inference

**Simple Approach**
- Remove $\forall$ and $\exists$
- Treat all first-order terms as simple symbols
- Solve using resolution for propositional logic

Example:

```
¬WarmBlooded(Platypus) ¬Mammal(Platypus) ∨ WarmBlooded(Platypus)
¬Mammal(Platypus) Mammal(Platypus)
false
```
Simple First-Order Inference

Simple Approach

- Remove $\forall$ and $\exists$
- Treat all first-order terms as simple symbols
- Solve using resolution for propositional logic

Example: $\text{WarmBlooded}(\text{Platypus})$?

\[
\begin{align*}
\neg \text{WarmBlooded}(\text{Platypus}) \\
\neg \text{Mammal}(\text{Platypus}) \lor \text{WarmBlooded}(\text{Platypus}) \\
\neg \text{Mammal}(\text{Platypus}) \\
\text{Mammal}(\text{Platypus}) \\
\hline
\text{false}
\end{align*}
\]
Simple First-Order Inference

Problem: Infinite Terms

- \textit{Mammal}(Steve)
- \textit{Mammal}(Mother(Steve))
- \textit{Mammal}(Mother(Mother(Steve)))
- \ldots

Solution: Iterative Deepening

Try proof with terms up to depth 1
Try proof with terms up to depth 2
\ldots
Proof found if exists, else infinite loop (semidecidable)
Simple First-Order Inference

Problem: Infinite Terms
- $Mammal(Steve)$
- $Mammal(Mother(Steve))$
- $Mammal(Mother(Mother(Steve)))$
- ...

Solution: Iterative Deepening
- Try proof with terms up to depth 1
- Try proof with terms up to depth 2
- ...

Proof found if exists, else infinite loop (semidecidable)
Simple First-Order Inference

Problem: Infinite Terms

- \textit{Mammal}(Steve)
- \textit{Mammal}(Mother(Steve))
- \textit{Mammal}(Mother(Mother(Steve))))
- ...

Solution: Iterative Deepening

- Try proof with terms up to depth 1
- Try proof with terms up to depth 2
- ...

Proof found if exists
Simple First-Order Inference

**Problem: Infinite Terms**
- $\text{Mammal}(\text{Steve})$
- $\text{Mammal}(\text{Mother}(\text{Steve}))$
- $\text{Mammal}(\text{Mother}(\text{Mother}(\text{Steve})))$
- ... 

**Solution: Iterative Deepening**
- Try proof with terms up to depth 1
- Try proof with terms up to depth 2
- ...

Proof found if exists, else infinite loop
Simple First-Order Inference

Problem: Infinite Terms

- Mammal(Steve)
- Mammal(Mother(Steve))
- Mammal(Mother(Mother(Steve)))
- ...

Solution: Iterative Deepening

- Try proof with terms up to depth 1
- Try proof with terms up to depth 2
- ...

Proof found if exists, else infinite loop (semidecidable)
Problems with Propositionalization

Prove: \textit{WarmBlooded}(\textit{Scooby})

\begin{align*}
\text{Dog}(\text{Scooby}) \\
\text{Dog}(\text{Scrappy}) \\
\forall x \; \text{Dog}(x) \Rightarrow \text{Mammal}(x) \\
\forall y \; \text{Mammal}(y) \Rightarrow \text{WarmBlooded}(y)
\end{align*}

Problem: Many Irrelevant Facts Produced

\begin{align*}
\text{Dog}(\text{Scrappy}) \\
\text{Dog}(\text{Scrappy}) \Rightarrow \text{Mammal}(\text{Scrappy}) \\
\text{Mammal}(\text{Scrappy}) \Rightarrow \text{WarmBlooded}(\text{Scrappy})
\end{align*}
Key Idea
Assign variables to make two expressions look the same

Formally
\[ \text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) \]
Unification

Key Idea
Assign variables to make two expressions look the same

Formally
\[ \text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) \]

Example: \( Eats(Panda, y)? \)

| \( p \) | \( q \) | \( \theta \) |
Unification

Key Idea
Assign variables to make two expressions look the same

Formally
\( \text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) \)

Example: \( \text{Eats}(\text{Panda}, y)? \)

\[
\begin{array}{ccc}
p & q & \theta \\
\hline
\text{Eats}(\text{Panda}, y) & \text{Eats}(\text{Panda}, \text{Leaves})
\end{array}
\]
Unification

Key Idea
Assign variables to make two expressions look the same

Formally
\[ \text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) \]

Example: \( \text{Eats}(\text{Panda}, y) \)?

\[
\begin{array}{c|c|c|c}
  p & q & \theta \\
  \hline
  \text{Eats}(\text{Panda}, y) & \text{Eats}(\text{Panda}, \text{Leaves}) & \{y=\text{Leaves}\}
\end{array}
\]
Unification

Key Idea
Assign variables to make two expressions look the same

Formally
\( \text{UNIFY}(p, q) = \theta \) where \( \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) \)

Example: \( \text{Eats}(Panda, y) \)?

\[
\begin{array}{c|c|c}
  p & q & \theta \\
  \hline
  \text{Eats}(Panda, y) & \text{Eats}(Panda, \text{Leaves}) & \{y=\text{Leaves}\} \\
  \text{Eats}(Panda, y) & \text{Eats}(x, \text{Pizza}) & \\
\end{array}
\]
Unification

Key Idea
Assign variables to make two expressions look the same

Formally
\[ \text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) \]

Example: \( Eats(Panda, y) \)?

\[
\begin{array}{ccc}
p & q & \theta \\
Eats(Panda, y) & Eats(Panda, Leaves) & \{y=Leaves\} \\
Eats(Panda, y) & Eats(x, Pizza) & \{x=Panda, y=Pizza\}
\end{array}
\]
### Unification

**Key Idea**

Assign variables to make two expressions look the same

**Formally**

\[
\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q)
\]

**Example: Eats(Panda, y)?**

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( \theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eats(Panda, y)</td>
<td>Eats(Panda, Leaves)</td>
<td>{y=Leaves}</td>
</tr>
<tr>
<td>Eats(Panda, y)</td>
<td>Eats(x, Pizza)</td>
<td>{x=Panda, y=Pizza}</td>
</tr>
<tr>
<td>Eats(Panda, y)</td>
<td>Eats(x, FavFood(x))</td>
<td></td>
</tr>
</tbody>
</table>
### Unification

**Key Idea**
Assign variables to make two expressions look the same

**Formally**

\[
\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q)
\]

**Example:** \(Eats(Panda, y)?\)

<table>
<thead>
<tr>
<th>(p)</th>
<th>(q)</th>
<th>(\theta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Eats(Panda, y))</td>
<td>(Eats(Panda, Leaves))</td>
<td>({y = \text{Leaves}})</td>
</tr>
<tr>
<td>(Eats(Panda, y))</td>
<td>(Eats(x, Pizza))</td>
<td>({x = \text{Panda}, y = \text{Pizza}})</td>
</tr>
<tr>
<td>(Eats(Panda, y))</td>
<td>(Eats(x, \text{FavFood}(x)))</td>
<td>({x = \text{Panda}, y = \text{FavFood}(\text{Panda})})</td>
</tr>
</tbody>
</table>
First Order Logic
Example Translations
First-Order Logic Inference
Inference Algorithms

Outline

1 First Order Logic
   - Motivation
   - Core Components
   - Quantifiers

2 Example Translations
   - Animal Kingdom
   - Wumpus World
   - Harry Potter

3 First-Order Logic Inference
   - Propositionalization
   - Unification

4 Inference Algorithms
   - Forward Chaining
   - Backward Chaining
   - Prolog
   - Resolution
**Generalized Modus Ponens**

**Definition**

\[
\begin{align*}
p_1' \\
p_2' \\
\vdots \\
p_n' \\
p_1 \land p_2 \land \ldots \land p_n & \Rightarrow q \\
\text{UNIFY}(p_1', p_1) &= \theta \\
\text{UNIFY}(p_2', p_2) &= \theta \\
\vdots \\
\text{UNIFY}(p_n', p_n) &= \theta \\
\text{SUBST}(\theta, q)
\end{align*}
\]
Generalized Modus Ponens

Definition

\[ p'_1 \]
\[ p'_2 \]
\[ \ldots \]
\[ p'_n \]
\[ p_1 \land p_2 \land \ldots \land p_n \Rightarrow q \]
\[ \text{UNIFY}(p'_1, p_1) = \theta \]
\[ \text{UNIFY}(p'_2, p_2) = \theta \]
\[ \ldots \]
\[ \text{UNIFY}(p'_n, p_n) = \theta \]
\[ \text{SUBST}(\theta, q) \]

Example

\[ \text{Odd}(17) \]
\[ \forall x \text{ Odd}(x) \Rightarrow \text{Mod}(x, 2, 1) \]
\[ \text{UNIFY} (\text{Odd}(17), \text{Odd}(x)) = \{ x = 17 \} \]
\[ \text{Mod}(17, 2, 1) \]
Forward Chaining

Key Ideas

- Repeatedly apply Generalized Modus Ponens
- Stop when nothing new can be inferred
Forward Chaining

Key Ideas
- Repeatedly apply Generalized Modus Ponens
- Stop when nothing new can be inferred

Details
KB must be only first-order definite clauses, one of:
- Atomic clauses, e.g. $\text{Mammal}(\text{Platypus})$
- Implications like $p_1 \land p_2 \land \ldots \land p_n \Rightarrow q$
**Forward Chaining Code**

```python
def forward_chaining(knowledge_base, query):
```

This code snippet demonstrates a forward chaining algorithm. The function `forward_chaining` takes a `knowledge_base` and a `query` as inputs. It iteratively adds new sentences to the knowledge base until no new sentences can be generated, and checks if the query can be concluded.
def forward_chaining(knowledge_base, query):
    # keep adding to the KB until no new sentences are generated
    new = True
    while sentences:
        new = set()
        for sentence in knowledge_base:
            # find something that unifies with the sentence body
            n = len(sentence.body)
            for subset in all_subsets(knowledge_base, n):
                assignment = unify(subset, sentence.body)
                if assignment is not None:
                    # conclude the sentence head
                    head = assignment.substitute(sentence.head)
                    if head not in knowledge_base + new:
                        new.add(head)
                    # return if the query is concluded
                    result = unify(head, query)
                    if result is not None:
                        return result
    knowledge_base.update(new)
def forward_chaining(knowledge_base, query):
    # keep adding to the KB until no new sentences are generated
    new = True
    while sentences:
        new = set()
        for sentence in knowledge_base:
            # find something that unifies with the sentence body
            n = len(sentence.body)
            for subset in all_subsets(knowledge_base, n):
                assignment = unify(subset, sentence.body)
                if assignment is not None:
                    # conclude the sentence head
                    head = assignment.substitute(sentence.head)
                    if head not in knowledge_base + new:
                        new.add(head)
                        # return if the query is concluded
                        result = unify(head, query)
                        if result is not None:
                            return result
        knowledge_base.update(new)
def forward_chaining(knowledge_base, query):
    # keep adding to the KB until no new sentences are generated
    new = True
    while sentences:
        new = set()
        for sentence in knowledge_base:
            # find something that unifies with the sentence body
            n = len(sentence.body)
            for subset in all_subsets(knowledge_base, n):
                assignment = unify(subset, sentence.body)
                if assignment is not None:
                    # conclude the sentence head
                    head = assignment.substitute(sentence.head)
                    if head not in knowledge_base + new:
                        new.add(head)

def unify(head, query):
    # return if the query is concluded
    result = unify(head, query)
    if result is not None:
        return result

knowledge_base.update(new)
def forward_chaining(knowledge_base, query):
    # keep adding to the KB until no new sentences are generated
    new = True
    while sentences:
        new = set()
        for sentence in knowledge_base:
            # find something that unifies with the sentence body
            n = len(sentence.body)
            for subset in all_subsets(knowledge_base, n):
                assignment = unify(subset, sentence.body)
                if assignment is not None:
                    # conclude the sentence head
                    head = assignment.substitute(sentence.head)
                    if head not in knowledge_base + new:
                        new.add(head)
                        # return if the query is concluded
                        result = unify(head, query)
                        if result is not None:
                            return result
        knowledge_base.update(new)
Forward Chaining Example

If you’re rich and someone sells something you want, you buy it
\[ \text{Rich}(x) \land \text{Wants}(x, y) \land \text{Sells}(z, y) \implies \text{Buys}(x, y) \]

If you’re hot, you want ice cream
\[ \text{Hot}(x) \land \text{IceCream}(y) \implies \text{Wants}(x, y) \]

Glacier sells all kinds of ice cream
\[ \text{IceCream}(y) \implies \text{Sells}(\text{Glacier}, y) \]

One flavor of ice cream is mint
\[ \text{IceCream}(	ext{Mint}) \]

Bill is rich
\[ \text{Rich}(\text{Bill}) \]

Bill is hot
\[ \text{Hot}(\text{Bill}) \]
Forward Chaining Example

\[
\begin{align*}
&\text{Hot}(Bill) \\
&\text{IceCream}(Mint) \\
&\text{Hot}(x) \land \text{IceCream}(y) \implies \text{Wants}(x, y) \\
&\text{Wants}(Bill, Mint)
\end{align*}
\]
Forward Chaining Example

\[
\begin{align*}
\text{Hot}(Bill) \\
\text{IceCream}(Mint) \\
\text{Hot}(x) \land \text{IceCream}(y) & \Rightarrow \text{Wants}(x, y) \\
\hline
\text{Wants}(Bill, Mint) \\
\text{IceCream}(Mint) \\
\text{IceCream}(y) & \Rightarrow \text{Sells}(Glacier, y) \\
\hline
\text{Sells}(Glacier, Mint)
\end{align*}
\]
Forward Chaining Example

1. \(\text{Hot}(\text{Bill})\)
2. \(\text{IceCream}(\text{Mint})\)
3. \(\text{Hot}(x) \land \text{IceCream}(y) \Rightarrow \text{Wants}(x, y)\)
4. \(\text{Wants}(\text{Bill}, \text{Mint})\)

5. \(\text{IceCream}(\text{Mint})\)
6. \(\text{IceCream}(y) \Rightarrow \text{Sells}(\text{Glacier}, y)\)
7. \(\text{Sells}(\text{Glacier}, \text{Mint})\)

8. \(\text{Rich}(\text{Bill})\)
9. \(\text{Wants}(\text{Bill}, \text{Mint})\)
10. \(\text{Sells}(\text{Glacier}, \text{Mint})\)
11. \(\text{Rich}(x) \land \text{Wants}(x, y) \land \text{Sells}(z, y) \Rightarrow \text{Buys}(x, y)\)
12. \(\text{Buys}(\text{Bill}, \text{Mint})\)
Forward Chaining Properties

Basic Properties

- Sound - uses Generalized Modus Ponens
- Complete - proof similar to propositional logic
- Works only with definite clauses
Forward Chaining Properties

Basic Properties
- Sound - uses Generalized Modus Ponens
- Complete - proof similar to propositional logic
- Works only with definite clauses

Termination
With no functions, $p$ predicates, $n$ constants, and at most $k$ arguments per predicate:
Forward Chaining Properties

Basic Properties
- Sound - uses Generalized Modus Ponens
- Complete - proof similar to propositional logic
- Works only with definite clauses

Termination
With no functions, $p$ predicates, $n$ constants, and at most $k$ arguments per predicate:

Maximum Facts:
Forward Chaining Properties

Basic Properties
- Sound - uses Generalized Modus Ponens
- Complete - proof similar to propositional logic
- Works only with definite clauses

Termination
With no functions, $p$ predicates, $n$ constants, and at most $k$ arguments per predicate:

Maximum Facts: $pn^k$
Forward Chaining Properties

Basic Properties
- Sound - uses Generalized Modus Ponens
- Complete - proof similar to propositional logic
- Works only with definite clauses

Termination
With no functions, $p$ predicates, $n$ constants, and at most $k$ arguments per predicate:

Maximum Facts: $pn^k$
Maximum Iterations:
Forward Chaining Properties

Basic Properties
- Sound - uses Generalized Modus Ponens
- Complete - proof similar to propositional logic
- Works only with definite clauses

Termination
With no functions, $p$ predicates, $n$ constants, and at most $k$ arguments per predicate:
- Maximum Facts: $pn^k$
- Maximum Iterations: $pn^k$
Forward Chaining Properties

**Basic Properties**
- Sound - uses Generalized Modus Ponens
- Complete - proof similar to propositional logic
- Works only with definite clauses

**Termination**
With no functions, $p$ predicates, $n$ constants, and at most $k$ arguments per predicate:

- **Maximum Facts:** $pn^k$
- **Maximum Iterations:** $pn^k$

With functions:
Forward Chaining Properties

Basic Properties
- Sound - uses Generalized Modus Ponens
- Complete - proof similar to propositional logic
- Works only with definite clauses

Termination
With no functions, $p$ predicates, $n$ constants, and at most $k$ arguments per predicate:
- Maximum Facts: $pn^k$
- Maximum Iterations: $pn^k$

With functions: may never terminate
Optimizing Forward Chaining

Indexing

- Treat facts like database relations
- Index by predicate + arguments
- Can get $O(1)$ fact retrieval
- Standard time/space tradeoffs
Indexing
- Treat facts like database relations
- Index by predicate + arguments
- Can get $O(1)$ fact retrieval
- Standard time/space tradeoffs

Rule Checking
- Don’t check all rules on each iteration
- Check rules when new part of premise is satisfied
Backward Chaining

Key Ideas
- Start with the terms in the query
- Look for sentences that can conclude those terms using Generalized Modus Ponens
- Recurse as necessary to find simple terms

Details
- KB must be only first-order definite clauses
def backward_chaining(knowledge_base, goals, assignment):
    if not goals:
        yield assignment
    else:
        term = assignment.substitute(goals[0])
        for sentence in knowledge_base:
            new_assignment = unify(sentence.head, term)
            if new_assignment is None:
                continue
            new_goals = list(sentence.body)
            new_goals.extend(goals[1:])
            new_assignment = new_assignment.compose(assignment)
            args = knowledge_base, new_goals, new_assignment
            for result in backward_chaining(*args):
                yield result
def backward_chaining(knowledge_base, goals, assignment):
    # if all goals have been resolved, generate the assignment
    if not goals:
        yield assignment
    else:
        term = assignment.substitute(goals[0])
        for sentence in knowledge_base:
            # try to unify the sentence with the first goal
            new_assignment = unify(sentence.head, term)
            if new_assignment is None:
                continue
            # add the sentence's premise to the remaining goals
            new_goals = list(sentence.body)
            new_goals.extend(goals[1:])
            new_assignment = new_assignment.compose(assignment)
            # recursively search for the remaining goals
            args = knowledge_base, new_goals, new_assignment
            for result in backward_chaining(*args):
                yield result
def backward_chaining(knowledge_base, goals, assignment):
    # if all goals have been resolved, generate the assignment
    if not goals:
        yield assignment
    # otherwise, check the first goal and then recurse
    else:
        term = assignment.substitute(goals[0])
        for sentence in knowledge_base:
            # try to unify the sentence with the first goal
            new_assignment = unify(sentence.head, term)
            if new_assignment is None:
                continue
            # add the sentence's premise to the remaining goals
            new_goals = list(sentence.body)
            new_goals.extend(goals[1:])
            new_assignment = new_assignment.compose(assignment)
            # recursively search for the remaining goals
            args = knowledge_base, new_goals, new_assignment
            for result in backward_chaining(*args):
                yield result
def backward_chaining(knowledge_base, goals, assignment):
    # if all goals have been resolved, generate the assignment
    if not goals:
        yield assignment
    # otherwise, check the first goal and then recurse
    else:
        term = assignment.substitute(goals[0])
        for sentence in knowledge_base:
            # try to unify the sentence with the first goal
            new_assignment = unify(sentence.head, term)
            if new_assignment is None:
                continue
            # add the sentence's premise to the remaining goals
            new_goals = list(sentence.body)
            new_goals.extend(goals[1:])
            new_assignment = new_assignment.compose(assignment)
            # recursively search for the remaining goals
            args = knowledge_base, new_goals, new_assignment
            for result in backward_chaining(*args):
                yield result
def backward_chaining(knowledge_base, goals, assignment):
    # if all goals have been resolved, generate the assignment
    if not goals:
        yield assignment
    # otherwise, check the first goal and then recurse
    else:
        term = assignment.substitute(goals[0])
        for sentence in knowledge_base:
            # try to unify the sentence with the first goal
            new_assignment = unify(sentence.head, term)
            if new_assignment is None:
                continue
            # add the sentence’s premise to the remaining goals
            new_goals = list(sentence.body)
            new_goals.extend(goals[1:])
            new_assignment = new_assignment.compose(assignment)
            for result in backward_chaining(*args):
                yield result
def backward_chaining(knowledge_base, goals, assignment):
    # if all goals have been resolved, generate the assignment
    if not goals:
        yield assignment
    # otherwise, check the first goal and then recurse
    else:
        term = assignment.substitute(goals[0])
        for sentence in knowledge_base:
            # try to unify the sentence with the first goal
            new_assignment = unify(sentence.head, term)
            if new_assignment is None:
                continue
            # add the sentence’s premise to the remaining goals
            new_goals = list(sentence.body)
            new_goals.extend(goals[1:])
            new_assignment = new_assignment.compose(assignment)
            # recursively search for the remaining goals
            args = knowledge_base, new_goals, new_assignment
            for result in backward_chaining(*args):
                yield result
Backward Chaining Example

Given:

\[ Rich(x) \land Wants(x, y) \land Sells(z, y) \Rightarrow Buys(x, y) \]
\[ Hot(x) \land IceCream(y) \Rightarrow Wants(x, y) \]
\[ IceCream(y) \Rightarrow Sells(Glacier, y) \]
\[ IceCream(Mint) \]
\[ Rich(Bill) \]
\[ Hot(Bill) \]

Query:

\[ Buys(x, y) \]
# Backward Chaining Example

<table>
<thead>
<tr>
<th>Goals</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ Buys(x, y) ]</td>
<td>{}</td>
</tr>
<tr>
<td>[ Rich(x), Wants(x, y), Sells(z, y) ]</td>
<td>{}</td>
</tr>
<tr>
<td>[ Wants(x, y), Sells(z, y) ]</td>
<td>{ x = Bill }</td>
</tr>
<tr>
<td>[ Hot(x), IceCream(y), Sells(z, y) ]</td>
<td>{ x = Bill }</td>
</tr>
<tr>
<td>[ IceCream(y), Sells(z, y) ]</td>
<td>{ x = Bill }</td>
</tr>
<tr>
<td>[ Sells(z, y) ]</td>
<td>{ x = Bill, y = Mint, z = Glacier }</td>
</tr>
<tr>
<td>[ IceCream(y) ]</td>
<td>{ x = Bill, y = Mint, z = Glacier }</td>
</tr>
</tbody>
</table>
### Backward Chaining Example

<table>
<thead>
<tr>
<th>Goals</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{Buys}(x, y)])</td>
<td>{}</td>
</tr>
</tbody>
</table>
### Backward Chaining Example

<table>
<thead>
<tr>
<th>Goals</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Buys(x, y)]</td>
<td>{}</td>
</tr>
<tr>
<td>[Rich(x), Wants(x, y), Sells(z, y)]</td>
<td>{}</td>
</tr>
</tbody>
</table>
## Backward Chaining Example

<table>
<thead>
<tr>
<th>Goals</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{Buys}(x, y)])</td>
<td>{}</td>
</tr>
<tr>
<td>([\text{Rich}(x), \text{Wants}(x, y), \text{Sells}(z, y)])</td>
<td>{}</td>
</tr>
<tr>
<td>([\text{Wants}(x, y), \text{Sells}(z, y)])</td>
<td>(x=\text{Bill})</td>
</tr>
</tbody>
</table>
## Backward Chaining Example

<table>
<thead>
<tr>
<th>Goals</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{Buys}(x, y)])</td>
<td>{}</td>
</tr>
<tr>
<td>([\text{Rich}(x), \text{Wants}(x, y), \text{Sells}(z, y)])</td>
<td>{}</td>
</tr>
<tr>
<td>([\text{Wants}(x, y), \text{Sells}(z, y)])</td>
<td>({x=\text{Bill}})</td>
</tr>
<tr>
<td>([\text{Hot}(x), \text{IceCream}(y), \text{Sells}(z, y)])</td>
<td>({x=\text{Bill}})</td>
</tr>
</tbody>
</table>
### Backward Chaining Example

<table>
<thead>
<tr>
<th>Goals</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[Buys(x, y)]$</td>
<td>${}$</td>
</tr>
<tr>
<td>$[Rich(x), Wants(x, y), Sells(z, y)]$</td>
<td>${}$</td>
</tr>
<tr>
<td>$[Wants(x, y), Sells(z, y)]$</td>
<td>${x=Bill}$</td>
</tr>
<tr>
<td>$[Hot(x), IceCream(y), Sells(z, y)]$</td>
<td>${x=Bill}$</td>
</tr>
<tr>
<td>$[IceCream(y), Sells(z, y)]$</td>
<td>${x=Bill}$</td>
</tr>
</tbody>
</table>
### Backward Chaining Example

<table>
<thead>
<tr>
<th>Goals</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[\text{Buys}(x, y)]$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$[\text{Rich}(x), \text{Wants}(x, y), \text{Sells}(z, y)]$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$[\text{Wants}(x, y), \text{Sells}(z, y)]$</td>
<td>${x=\text{Bill}}$</td>
</tr>
<tr>
<td>$[\text{Hot}(x), \text{IceCream}(y), \text{Sells}(z, y)]$</td>
<td>${x=\text{Bill}}$</td>
</tr>
<tr>
<td>$[\text{IceCream}(y), \text{Sells}(z, y)]$</td>
<td>${x=\text{Bill}}$</td>
</tr>
<tr>
<td>$[\text{Sells}(z, y)]$</td>
<td>${x=\text{Bill}, y=\text{Mint}}$</td>
</tr>
</tbody>
</table>
Backward Chaining Example

Goals

\[ Buys(x, y) \]
\[ Rich(x), Wants(x, y), Sells(z, y) \]
\[ Wants(x, y), Sells(z, y) \]
\[ Hot(x), IceCream(y), Sells(z, y) \]
\[ IceCream(y), Sells(z, y) \]
\[ Sells(z, y) \]
\[ IceCream(y) \]

Assignment

\{ \}
\{ \}
\{ x=Bill \}
\{ x=Bill \}
\{ x=Bill \}
\{ x=Bill, y=Mint \}
\{ x=Bill, y=Mint, z=Glacier \}
### Backward Chaining Example

<table>
<thead>
<tr>
<th>Goals</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Buys(x, y)]</td>
<td>{}</td>
</tr>
<tr>
<td>[Rich(x), Wants(x, y), Sells(z, y)]</td>
<td>{}</td>
</tr>
<tr>
<td>[Wants(x, y), Sells(z, y)]</td>
<td>{x=\text{Bill}}</td>
</tr>
<tr>
<td>[Hot(x), IceCream(y), Sells(z, y)]</td>
<td>{x=\text{Bill}}</td>
</tr>
<tr>
<td>[IceCream(y), Sells(z, y)]</td>
<td>{x=\text{Bill}}</td>
</tr>
<tr>
<td>[Sells(z, y)]</td>
<td>{x=\text{Bill}, y=\text{Mint}}</td>
</tr>
<tr>
<td>[IceCream(y)]</td>
<td>{x=\text{Bill}, y=\text{Mint}, z=\text{Glacier}}</td>
</tr>
<tr>
<td>[]</td>
<td>{x=\text{Bill}, y=\text{Mint}, z=\text{Glacier}}</td>
</tr>
</tbody>
</table>
Backward Chaining Properties

Properties

- **Sound**: Yes, uses Generalized Modus Ponens
- **Complete**: No, could hit infinite loops
  - Fix by keeping track of goals already seen
- **Space**: Linear in size of proof (depth-first search)
Backward Chaining Properties

### Properties

**Sound**

- Yes, uses Generalized Modus Ponens
- Complete: No, could hit infinite loops
  - Fix by keeping track of goals already seen
- Space: Linear in size of proof (depth-first search)
# Backward Chaining Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td>Yes, uses Generalized Modus Ponens</td>
</tr>
</tbody>
</table>

- **Sound**:
  - Yes, uses Generalized Modus Ponens
  - Complete: No, could hit infinite loops
  - Space: Linear in size of proof (depth-first search)
Properties

**Sound**
Yes, uses Generalized Modus Ponens

**Complete**
No, could hit infinite loops
⇒ fix by keeping track of goals already seen

**Space**
Linear in size of proof (depth-first search)
### Backward Chaining Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td>Yes, uses Generalized Modus Ponens</td>
</tr>
<tr>
<td><strong>Complete</strong></td>
<td>No, could hit infinite loops</td>
</tr>
</tbody>
</table>
Backward Chaining Properties

### Properties

- **Sound**: Yes, uses Generalized Modus Ponens
- **Complete**: No, could hit infinite loops
  - ⇒ fix by keeping track of goals already seen
## Backward Chaining Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td>Yes, uses Generalized Modus Ponens</td>
</tr>
</tbody>
</table>
| **Complete** | No, could hit infinite loops  
  ⇒ fix by keeping track of goals already seen |
| **Space**  | Linear in size of proof (depth-first search) |
## Backward Chaining Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td>Yes, uses Generalized Modus Ponens</td>
</tr>
<tr>
<td><strong>Complete</strong></td>
<td>No, could hit infinite loops. $\Rightarrow$ fix by keeping track of goals already seen</td>
</tr>
<tr>
<td><strong>Space</strong></td>
<td>Linear in size of proof (depth-first search)</td>
</tr>
</tbody>
</table>
Prolog

Prolog Overview

- Backward chaining + many optimizations
- Millions of logical inferences per second
## Prolog

### Prolog Overview
- Backward chaining + many optimizations
- Millions of logical inferences per second

### Prolog Syntax
- Head first, then body
- Predicates lowercase, variables uppercase
- Comma for $\land$, semicolon for $\lor$

```prolog
factorial(1, 1).
factorial(N, _) :- N <= 0, fail.
factorial(N, F) :- N > 1, N1 is N - 1,
    factorial(N1, F1), F is F1 * N.
```
Prolog Examples
Prove: \( \text{Connected}(A, F) \)

Given: \( \forall x, z \ Edge(x, z) \Rightarrow \text{Connected}(x, z) \) 
\[ \forall x, y, z \ Edge(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z) \] 
\[ \text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F) \]
Prove: $\text{Connected}(A, F)$

Given: $\forall x, z \ \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z)$

$\forall x, y, z \ \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z)$

$\text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F)$

Diagram: Connected($A, F$)
Prove: Connected(A, F)

Given: \( \forall x, z \, \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z) \)
\[ \forall x, y, z \, \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z) \]
\[ \text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F) \]
Prove: \( \text{Connected}(A, F) \)

Given:
\[
\forall x, z \ \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z)
\]
\[
\forall x, y, z \ \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z)
\]
\[
\text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F)
\]

\[
\text{Connected}(A, F)
\]

\[
\text{Edge}(A, F)
\]

\[
\text{fail}
\]
Prove: \( \text{Connected}(A, F) \)

Given: \( \forall x, z \, \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z) \)

\( \forall x, y, z \, \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z) \)

\( \text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F) \)

Connected(A, F)
Prove: \( \text{Connected}(A, F) \)

Given:

\[ \forall x, z \, \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z) \]
\[ \forall x, y, z \, \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z) \]
\[ \text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F) \]
Prove: $\text{Connected}(A, F)$

Given: $\forall x, z \; \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z)$

$\forall x, y, z \; \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z)$

$\text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F)$

Diagram:

```
  Connected(A, F)
     /    \
    /      \
  Edge(A, y)  Connected(y, F)
     /      \
    /        \
  Edge(A, B)  \
```

$\text{Edge}(A, y)$

$\text{Edge}(A, B)$
Backwards Chaining Backtracking

Prove: \( \text{Connected}(A, F) \)

Given:

\[ \forall x, z \; \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z) \]
\[ \forall x, y, z \; \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z) \]
\[ \text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F) \]

Diagram:

```
Connected(A, F)
  /\  \
Edge(A, B)
  /\  \
Edge(A, B)
```

```
Connected(B, F)
```

```
```
Prove: $\text{Connected}(A, F)$

Given:

$\forall x, z \ \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z)$

$\forall x, y, z \ \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z)$

$\text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F)$

Diagram:

```
Connected(A, F)
   /         \
Edge(A, B)   Connected(B, F)
   /         \
Edge(A, B)  Edge(B, y)  Connected(y, F)
```
Prove: $\text{Connected}(A, F)$

Given: $\forall x, z \ \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z)$

$\forall x, y, z \ \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z)$

$\text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F)$
Prove: \( \text{Connected}(A, F) \)

Given: \( \forall x, z \ \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z) \)

\( \forall x, y, z \ \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z) \)

\( \text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F) \)
Prove: $\text{Connected}(A, F)$

Given: $\forall x, z \text{ Edge}(x, z) \Rightarrow \text{Connected}(x, z)$

$\forall x, y, z \text{ Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z)$

$\text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F)$

$\text{Connected}(A, F)$

\begin{align*}
\text{Edge}(A, B) & \quad \text{Connected}(B, F) \\
\text{Edge}(A, B) & \quad \text{Connected}(C, F) \\
\text{Edge}(B, C) & \quad \text{Connected}(C, F) \quad \text{... fail ...}
\end{align*}
Prove: \( \text{Connected}(A, F) \)

Given: \( \forall x, z \ Edge(x, z) \Rightarrow \text{Connected}(x, z) \)
\[
\forall x, y, z \ Edge(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z)
\]
\[
\text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F)
\]
Backwards Chaining Backtracking

Prove: $\text{Connected}(A, F)$

Given:

1. $\forall x, z \ (\text{Edge}(x, z) \Rightarrow \text{Connected}(x, z))$
2. $\forall x, y, z \ (\text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z))$
3. $\text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F)$

The proof is shown in the diagram below:

- Start with $\text{Connected}(A, F)$.
- Move down the left branch with $\text{Edge}(A, B)$.
- Move down the right branch with $\text{Connected}(B, F)$.
- Further down with $\text{Edge}(B, D)$.
- Finally, reach $\text{Connected}(y, F)$ with $\text{Edge}(B, y)$.

This verifies the proof of $\text{Connected}(A, F)$.
Prove: \( \text{Connected}(A, F) \)

Given: \( \forall x, z \ \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z) \)
\[
\forall x, y, z \ \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z)
\]
\[
\text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F)
\]
Prove: $\text{Connected}(A, F)$

Given:

$\forall x, z \; \text{Edge}(x, z) \Rightarrow \text{Connected}(x, z)$

$\forall x, y, z \; \text{Edge}(x, y) \land \text{Connected}(y, z) \Rightarrow \text{Connected}(x, z)$

$\text{Edge}(A, B) \land \text{Edge}(B, C) \land \text{Edge}(B, D) \land \text{Edge}(D, F)$
Prolog Example
Resolution

**Definition**

\[
\begin{align*}
p_1 \lor \ldots \lor p_n \\
q_1 \lor \ldots \lor q_m \\
\text{UNIFY}(p_i, \neg q_j) = \theta \\
\text{SUBST}(\theta, p_1 \lor \ldots p_{i-1} \lor p_{i+1} \ldots \lor p_n \\
\lor q_1 \lor \ldots q_{j-1} \lor q_{j+1} \ldots \lor q_m)
\end{align*}
\]
Resolution

**Definition**

\[
p_1 \lor \ldots \lor p_n \\
q_1 \lor \ldots \lor q_m \\
\text{UNIFY}(p_i, \neg q_j) = \theta \\
\text{SUBST}(\theta, p_1 \lor \ldots p_{i-1} \lor p_{i+1} \ldots \lor p_n \\
\lor q_1 \lor \ldots q_{j-1} \lor q_{j+1} \ldots \lor q_m)
\]

**Example**

\[
\neg Mammal(x) \lor WarmBlooded(x) \\
Mammal(\text{Platypus}) \\
\hline
WarmBlooded(\text{Platypus})
\]
Resolution

Resolution Procedure

1. Convert knowledge base to CNF
2. Convert query to CNF
3. Assume $\neg$ query
4. Apply resolution until $false$ is concluded
Resolution

Resolution Procedure

1. Convert knowledge base to CNF
2. Convert query to CNF
3. Assume $\neg$ query
4. Apply resolution until $false$ is concluded

CNF Complications

- Negations moved through $\forall$ and $\exists$
- All quantifiers must have different variable names
- Quantifier scopes handled through skolemization
Resolution Example

Given:
\[ \neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \lor Buys(x, y) \]
\[ \neg Hot(x) \lor \neg IceCream(y) \lor Wants(x, y) \]
\[ \neg IceCream(y) \lor Sells(Glacier, y) \]
\[ IceCream(Mint) \]
\[ Rich(Bill) \]
\[ Hot(Bill) \]

Prove:
\[ Buys(x, y) \]
Resolution Example

\[ \neg \text{Buys}(x, y) \]
Resolution Example

\[ \neg \text{Buys}(x, y) \]
\[ \neg \text{Rich}(x) \lor \neg \text{Wants}(x, y) \lor \neg \text{Sells}(z, y) \lor \text{Buys}(x, y) \]
Resolution Example

\[ \neg Buys(x, y) \]
\[ \neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \lor Buys(x, y) \]

\[ \neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \]
Resolution Example

\[ \neg Buys(x, y) \]
\[ \neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \lor Buys(x, y) \]
\[ \neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \]
\[ Rich(Bill) \]
Resolution Example

\[\neg Buys(x, y)\]
\[\neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \lor Buys(x, y)\]
\[\neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y)\]
\[Rich(Bill)\]
\[\neg Wants(Bill, y) \lor \neg Sells(z, y)\]
Resolution Example

\[
\neg Buys(x, y) \\
\neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \lor Buys(x, y) \\
\neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \\
Rich(Bill) \\
\neg Wants(Bill, y) \lor \neg Sells(z, y) \\
\neg Hot(x) \lor \neg IceCream(y) \lor Wants(x, y)
\]
Resolution Example

\[ \neg Buys(x, y) \]
\[ \neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \lor Buys(x, y) \]
\[ \neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \]
\[ Rich(Bill) \]
\[ \neg Wants(Bill, y) \lor \neg Sells(z, y) \]
\[ \neg Hot(x) \lor \neg IceCream(y) \lor Wants(x, y) \]
\[ \neg Hot(Bill) \lor \neg IceCream(y) \lor \neg Sells(z, y) \]
Resolution Example

\[\neg Buys(x, y)\]
\[\neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \lor Buys(x, y)\]
\[\neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y)\]
\[Rich(Bill)\]
\[\neg Wants(Bill, y) \lor \neg Sells(z, y)\]
\[\neg Hot(x) \lor \neg IceCream(y) \lor Wants(x, y)\]
\[\neg Hot(Bill) \lor \neg IceCream(y) \lor \neg Sells(z, y)\]
\[\neg IceCream(y) \lor Sells(Glacier, y)\]
Resolution Example

\[ \neg Buys(x, y) \]
\[ \neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \lor Buys(x, y) \]
\[ \neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \]
\[ Rich(Bill) \]
\[ \neg Wants(Bill, y) \lor \neg Sells(z, y) \]
\[ \neg Hot(x) \lor \neg IceCream(y) \lor Wants(x, y) \]
\[ \neg Hot(Bill) \lor \neg IceCream(y) \lor \neg Sells(z, y) \]
\[ \neg IceCream(y) \lor Sells(Glacier, y) \]
\[ \neg Hot(Bill) \lor \neg IceCream(y) \]
Resolution Example

\(-\text{Buys}(x, y)\)
\(-\text{Rich}(x) \lor -\text{Wants}(x, y) \lor -\text{Sells}(z, y) \lor \text{Buys}(x, y)\)
\(-\text{Rich}(x) \lor -\text{Wants}(x, y) \lor -\text{Sells}(z, y)\)
\text{Rich}(\text{Bill})

\(-\text{Wants}(\text{Bill}, y) \lor -\text{Sells}(z, y)\)
\(-\text{Hot}(x) \lor -\text{IceCream}(y) \lor \text{Wants}(x, y)\)
\(-\text{Hot}(\text{Bill}) \lor -\text{IceCream}(y) \lor -\text{Sells}(z, y)\)
\(-\text{IceCream}(y) \lor \text{Sells}(\text{Glacier}, y)\)
\(-\text{Hot}(\text{Bill}) \lor -\text{IceCream}(y)\)
\text{IceCream}(\text{Mint})
Resolution Example

\[ \neg Buys(x, y) \]
\[ \neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \lor Buys(x, y) \]
\[ \neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \]
\[ Rich(Bill) \]
\[ \neg Wants(Bill, y) \lor \neg Sells(z, y) \]
\[ \neg Hot(x) \lor \neg IceCream(y) \lor Wants(x, y) \]
\[ \neg Hot(Bill) \lor \neg IceCream(y) \lor \neg Sells(z, y) \]
\[ \neg IceCream(y) \lor Sells(Glacier, y) \]
\[ \neg Hot(Bill) \lor \neg IceCream(y) \]
\[ IceCream(Mint) \]
\[ \neg Hot(Bill) \]
Resolution Example

\[-Buys(x, y)\]
\[-Rich(x) \lor -Wants(x, y) \lor -Sells(z, y) \lor Buys(x, y)\]
\[-Rich(x) \lor -Wants(x, y) \lor -Sells(z, y)\]
Rich(Bill)

\[-Wants(Bill, y) \lor -Sells(z, y)\]
\[-Hot(x) \lor -IceCream(y) \lor Wants(x, y)\]

\[-Hot(Bill) \lor -IceCream(y) \lor -Sells(z, y)\]
\[-IceCream(y) \lor Sells(Glacier, y)\]

\[-Hot(Bill) \lor -IceCream(y)\]
IceCream(Mint)

\[-Hot(Bill)\]
Hot(Bill)
Resolution Example

\[
\neg Buys(x, y) \\
\neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \lor Buys(x, y) \\
\neg Rich(x) \lor \neg Wants(x, y) \lor \neg Sells(z, y) \\
Rich(Bill) \\
\neg Wants(Bill, y) \lor \neg Sells(z, y) \\
\neg Hot(x) \lor \neg IceCream(y) \lor Wants(x, y) \\
\neg Hot(Bill) \lor \neg IceCream(y) \lor \neg Sells(z, y) \\
\neg IceCream(y) \lor Sells(Glacier, y) \\
\neg Hot(Bill) \lor \neg IceCream(y) \\
IceCream(Mint) \\
\neg Hot(Bill) \\
Hot(Bill) \\
false
\]
Resolution Chaining Properties

Properties

- **Sound**
  - Yes, uses Resolution
  - Complete with factoring to combine unifiable literals
  - Requires extra work to retrieve substitutions
  - Can produce useless answers for existential goals
Resolution Chaining Properties

Properties

**Sound**
Yes, uses Resolution
## Resolution Chaining Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sound</td>
<td>Yes, uses Resolution</td>
</tr>
<tr>
<td>Complete</td>
<td>Yes, with factoring to combine unifiable literals</td>
</tr>
<tr>
<td></td>
<td>Requires extra work to retrieve substitutions</td>
</tr>
<tr>
<td></td>
<td>Can produce useless answers for existential goals</td>
</tr>
</tbody>
</table>
## Resolution Chaining Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td>Yes, uses Resolution</td>
</tr>
</tbody>
</table>
| **Complete**   | No, if only binary resolution is used  
             Yes, with factoring to combine unifiable literals |
### Resolution Chaining Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td>Yes, uses Resolution</td>
</tr>
<tr>
<td><strong>Complete</strong></td>
<td>No, if only binary resolution is used</td>
</tr>
<tr>
<td></td>
<td>Yes, with factoring to combine unifiable literals</td>
</tr>
</tbody>
</table>

### Other Properties

- Requires extra work to retrieve substitutions
- Can produce useless answers for existential goals
Key Ideas

Translating First-Order Logic

- Identify objects (terms) and relations (predicates)
- Generally, use ⇒ with ∀ and ∧ with ∃
- Use inequality to specify unique objects

First-Order Logic Inference

- Forward chaining on definite clauses is sound and complete
- Backward chaining on definite clauses is sound
- Resolution is sound and complete