Using First-Order Logic (FOL)

Outline

I. Numbers and sets in FOL

II. The wumpus world in FOL

III. Knowledge engineering process

IV. Instantiation and Skolemization

V. Unification

* Figures are from the textbook site unless a source is specifically cited.
Describe the theory of natural numbers using merely:

- one constant symbol, 0
- one function symbol, $S$ (successor)
- one predicate, $NatNum$
I. Domain of Natural Numbers

Describe the theory of natural numbers using merely:

- one constant symbol, 0
- one function symbol, $S$ (successor)
- one predicate, $NatNum$

✿ Recursive definition:

\[
NatNum(0) \\
\forall n \ NatNum(n) \Rightarrow NatNum(S(n))
\]
I. Domain of Natural Numbers

Describe the theory of natural numbers using merely:

- one constant symbol, 0
- one function symbol, \( S \) (successor)
- one predicate, \( \text{NatNum} \)

♦ Recursive definition:

\[
\text{NatNum}(0) \\
\forall n \; \text{NatNum}(n) \rightarrow \text{NatNum}(S(n))
\]

♦ Axioms to constrain the successor function:

\[
\forall n \; 0 \neq S(n) \\
\forall m, n \; m \neq n \Rightarrow S(m) \neq S(n)
\]
∀m NatNum(m) ⇒ +(0, m) = m
Defining Addition

∀m NatNum(m) ⇒ +(0, m) = m

prefix notation
Defining Addition

∀m NatNum(m) ⇒ +(0, m) = m

prefix notation

∀m NatNum(m) ⇒ 0 + m = m

Use infix notation for readability.
Defining Addition

∀m NatNum(m) ⇒ +(0, m) = m

prefix notation

∀m NatNum(m) ⇒ 0 + m = m

infix notation

Use infix notation for readability.
Defining Addition

∀ \( m \) NatNum\( (m) \) ⇒ \(+ (0, m) = m\)

prefix notation

∀ \( m \) NatNum\( (m) \) ⇒ \( 0 + m = m \)

infix notation

∀ \( m, n \) NatNum\( (m) \) ∨ NatNum\( (n) \) ⇒ \(+ (S(m), n) = S(+ (m, n))\)

Use infix notation for readability.
Defining Addition

∀ 𝑚 NatNum(𝑚) ⇒ + (0, 𝑚) = 𝑚

prefix notation

∀ 𝑚 NatNum(𝑚) ⇒ 0 + 𝑚 = 𝑚

infix notation

∀ 𝑚, 𝑛 NatNum(𝑚) ∧ NatNum(𝑛) ⇒ +(𝑆(𝑚), 𝑛) = 𝑆(+(𝑚, 𝑛))

Write 𝑆(𝑛) as 𝑛 + 1.

∀ 𝑚, 𝑛 NatNum(𝑚) ∧ NatNum(𝑛) ⇒ (𝑚 + 1) + 𝑛 = (𝑚 + 𝑛) + 1
Defining Addition

∀m NatNum(m) ⇒ +(0, m) = m

prefix notation

∀m NatNum(m) ⇒ 0 + m = m

infix notation

∀m, n NatNum(m) ∧ NatNum(n) ⇒ +(S(m), n) = S(+m, n))

Syntactic sugar: an extension to the standard syntax that does not change the semantics but improves readability.

∀m, n NatNum(m) ∧ NatNum(n) ⇒ (m + 1) + n = (m + n) + 1

Write S(n) as n + 1.
Defining Addition

∀ \( m \) \( \text{NatNum}(m) \) ⇒ \( +(0, m) = m \)

(prefix notation)

∀ \( m \) \( \text{NatNum}(m) \) ⇒ \( 0 + m = m \)

(infix notation)

∀ \( m, n \) \( \text{NatNum}(m) \land \text{NatNum}(n) \) ⇒ \( +(S(m), n) = S(+m, n)) \)

Syntactic sugar: an extension to the standard syntax that does not change the semantics but improves readability.

Write \( S(n) \) as \( n + 1 \).

∀ \( m, n \) \( \text{NatNum}(m) \land \text{NatNum}(n) \) ⇒ \( (m + 1) + n = (m + n) + 1 \)

Syntactic sugar: \( \forall m \forall n \)
according to syntax
Domain of Sets

Syntactic sugar:

- \{\} for the empty set
- One unary predicate, \textit{Set}
Domain of Sets

Syntatic sugar:

- {} for the empty set
- One unary predicate, Set
- Binary predicate, ∈

  e.g., $x \in s$ ($x$ is a member of set $s$)
Domain of Sets

Syntactic sugar:

- \{\} for the empty set
- One unary predicate, \textit{Set}
- Binary predicate, \(\in\)
  
  e.g., \(x \in s\) (\(x\) is a member of set \(s\))
- Binary predicate, \(\subseteq\)
  
  e.g., \(s_1 \subseteq s_2\) (set \(s_1\) is a subset of set \(s_2\))
Domain of Sets

Syntactic sugar:

- \{\} for the empty set
- One unary predicate, \textit{Set}
- Binary predicate, \(\in\)
  
e.g., \(x \in s\) (\(x\) is a member of set \(s\))
- Binary predicate, \(\subseteq\)
  
e.g., \(s_1 \subseteq s_2\) (set \(s_1\) is a subset of set \(s_2\))
- Binary functions, \(\cap\) (intersection), \(\cup\) (union), \textit{Add}
  
  \(s_1 \cap s_2\)
Domain of Sets

Syntactic sugar:

- \{\} for the empty set
- One unary predicate, $\text{Set}$
- Binary predicate, $\in$
  
  e.g., $x \in s$ ($x$ is a member of set $s$)
- Binary predicate, $\subseteq$
  
  e.g., $s_1 \subseteq s_2$ (set $s_1$ is a subset of set $s_2$)
- Binary functions, $\cap$ (intersection), $\cup$ (union), $\text{Add}$
  
  $s_1 \cap s_2$, $s_1 \cup s_2$
Domain of Sets

Syntactic sugar:

- \{\} for the empty set
- One unary predicate, \(\text{Set}\)
- Binary predicate, \(\in\)
  - e.g., \(x \in s\) (\(x\) is a member of set \(s\))
- Binary predicate, \(\subseteq\)
  - e.g., \(s_1 \subseteq s_2\) (set \(s_1\) is a subset of set \(s_2\))
- Binary functions, \(\cap\) (intersection), \(\cup\) (union), \(\text{Add}\)
  - \(s_1 \cap s_2\)
  - \(s_1 \cup s_2\)
  - \(\text{Add}(x, s)\)
    - the set resulting from add element \(x\) to set \(s\)
Eight Axioms for Sets

A set is either an empty set or made by adding something to a set.

\[ \forall s \; \text{Set}(s) \iff (s = \{\}) \lor (\exists x, s_0 \; \text{Set}(s_0) \land s = \text{Add}(x, s_0)) \]
Eight Axioms for Sets

• A set is either an empty set or made by adding something to a set.

\[ \forall s \; \text{Set}(s) \iff (s = \emptyset) \lor (\exists x, s_0 \; \text{Set}(s_0) \land s = \text{Add}(x, s_0)) \]

• The empty set has no elements added to it.

\[ \neg \exists x, s \; \text{Add}(x, s) = \emptyset \]  
// equivalently, no way to decompose \{\} 
// into a smaller set and an element
Eight Axioms for Sets

- A set is either an empty set or made by adding something to a set.

\[ \forall s \ Set(s) \iff (s = \{\}) \lor (\exists x, s_0 \ Set(s_0) \land s = Add(x, s_0)) \]

- The empty set has no elements added to it.

\[ \neg \exists x, s \ Add(x, s) = \{\} \]  // equivalently, no way to decompose {} into a smaller set and an element

- Adding an element already in the set has no effect.

\[ \forall x, s \ x \in s \iff s = Add(x, s) \]
Eight Axioms for Sets

- A set is either an empty set or made by adding something to a set.

\[ \forall s \; \text{Set}(s) \iff (s = \{\}) \lor (\exists x, s_0 \; \text{Set}(s_0) \land s = \text{Add}(x, s_0)) \]

- The empty set has no elements added to it.

\[ \neg \exists x, s \; \text{Add}(x, s) = \{\} \]

// equivalently, no way to decompose {} into a smaller set and an element

- Adding an element already in the set has no effect.

\[ \forall x, s \; x \in s \iff s = \text{Add}(x, s) \]

- The only members of set are the added elements.
Eight Axioms for Sets

♦ A set is either an empty set or made by adding something to a set.

\[ \forall s \ Set(s) \leftrightarrow (s = \{\}) \lor (\exists x, s_0 \ Set(s_0) \land s = Add(x, s_0)) \]

♦ The empty set has no elements added to it.

\[ \neg \exists x, s \ Add(x, s) = \{\} \quad // \text{equivalently, no way to decompose} \{\} \quad // \text{into a smaller set and an element} \]

♦ Adding an element already in the set has no effect.

\[ \forall x, s \ x \in s \leftrightarrow s = Add(x, s) \]

♦ The only members of set are the added elements.

\[ \forall x, s \ x \in s \leftrightarrow \exists y, s_2 \ (s = Add(y, s_2) \land (x = y \lor x \in s_2)) \]

// expressed recursively
Eight Axioms for Sets

♦ A set is either an empty set or made by adding something to a set.

\[ \forall s \ Set(s) \iff (s = \{\}) \lor (\exists x, s_0 \ Set(s_0) \land s = \text{Add}(x, s_0)) \]

♦ The empty set has no elements added to it.

\[ \neg \exists x, s \ Add(x, s) = \{\} \]

// equivalently, no way to decompose {} into a smaller set and an element

♦ Adding an element already in the set has no effect.

\[ \forall x, s \ x \in s \iff s = \text{Add}(x, s) \]

♦ The only members of set are the added elements.

\[ \forall x, s \ x \in s \iff \exists y, s_2 \ (s = \text{Add}(y, s_2) \land (x = y \lor x \in s_2)) \]

// expressed recursively

must be true at some recursion level
Set Axioms (cont’d)

♦ Subset relationship

\[ \forall s_1, s_2 \quad s_1 \subseteq s_2 \iff (\forall x \ x \in s_1 \Rightarrow x \in s_2) \]
Set Axioms (cont’d)

♦ Subset relationship

\[ \forall s_1, s_2 \quad s_1 \subseteq s_2 \iff (\forall x \ x \in s_1 \Rightarrow x \in s_2) \]

♦ Set equality

\[ \forall s_1, s_2 \quad s_1 = s_2 \iff (s_1 \subseteq s_2 \land s_2 \subseteq s_1) \]
Set Axioms (cont’d)

♦ Subset relationship

\[ \forall s_1, s_2 \quad s_1 \subseteq s_2 \iff (\forall x \ x \in s_1 \Rightarrow x \in s_2) \]

♦ Set equality

\[ \forall s_1, s_2 \quad s_1 = s_2 \iff (s_1 \subseteq s_2 \land s_2 \subseteq s_1) \]

♦ Intersection

\[ \forall x, s_1, s_2 \quad x \in s_1 \cap s_2 \iff (x \in s_1 \land x \in s_2) \]
Set Axioms (cont’d)

- **Subset relationship**
  \[
  \forall s_1, s_2 \quad s_1 \subseteq s_2 \iff (\forall x \ x \in s_1 \Rightarrow x \in s_2)
  \]

- **Set equality**
  \[
  \forall s_1, s_2 \quad s_1 = s_2 \iff (s_1 \subseteq s_2 \land s_2 \subseteq s_1)
  \]

- **Intersection**
  \[
  \forall x, s_1, s_2 \quad x \in s_1 \cap s_2 \iff (x \in s_1 \land x \in s_2)
  \]

- **Union**
  \[
  \forall x, s_1, s_2 \quad x \in s_1 \cup s_2 \iff (x \in s_1 \lor x \in s_2)
  \]
II. The Wumpus World in FOL

First-order logic axioms are much more concise than propositional axioms.
First-order logic axioms are much more concise than propositional axioms.

- Predicate for percept

\[ \text{Percept}([\text{Stench, Breeze, Glitter, None, None}], 5) \]
II. The Wumpus World in FOL

First-order logic axioms are much more concise than propositional axioms.

- Predicate for percept

\[ \text{Percept}([\text{Stench}, \text{Breeze}, \text{Glitter}, \text{None}, \text{None}], 5) \]

sensor reading

No Bump  No Scream
II. The Wumpus World in FOL

First-order logic axioms are much more concise than propositional axioms.

- Predicate for percept

\[ \text{Percept}([\text{Stench}, \text{Breeze}, \text{Glitter}, \text{None}, \text{None}], 5) \]
II. The Wumpus World in FOL

First-order logic axioms are much more concise than propositional axioms.

- Predicate for percept
  \[ \text{Percept}([\text{Stench, Breeze, Glitter, None, None}], 5) \]

- Actions
  \[ \text{Turn(Right), Turn(Left), Forward, Shoot, Grab, Climb} \]

\[ \text{No Bump, No Scream} \]

\[ \text{sensor reading, time} \]
II. The Wumpus World in FOL

First-order logic axioms are much more concise than propositional axioms.

- Predicate for percept
  \[ \text{Percept}([\text{Stench}, \text{Breeze}, \text{Glitter}, \text{None}, \text{None}], 5) \]

- Actions
  \[ \text{Turn}(\text{Right}), \text{Turn}(\text{Left}), \text{Forward}, \text{Shoot}, \text{Grab}, \text{Climb} \]

- Querying the KB for best action
  \[ \text{ASKVARS}(KB, \text{BestAction}(a, 5)) \]

A binding list, e.g., \{a/Grab\}, is returned.
No need for the use of fluents (e.g., $Forward^t$, $Bump^{t+1}$)!

\[
\forall t, s, g, w, c \text{ Percept}([s, \text{Breeze}, g, w, c], t) \Rightarrow \text{Breeze}(t)
\]

\[
\forall t, s, g, w, c \text{ Percept}([s, \text{None}, g, w, c], t) \Rightarrow \neg \text{Breeze}(t)
\]

\[
\forall t, s, b, w, c \text{ Percept}([s, b, \text{Glitter}, w, c], t) \Rightarrow \text{Glitter}(t)
\]

\[
\forall t, s, b, w, c \text{ Percept}([s, b, \text{None}, w, c], t) \Rightarrow \neg \text{Glitter}(t)
\]

\[
\vdots
\]
No need for the use of fluents (e.g., $Forward^t$, $Bump^{t+1}$)!

∀ $t, s, g, w, c$  $\text{Percept}([s, \text{Breeze}, g, w, c], t) \Rightarrow \text{Breeze}(t)$

∀ $t, s, g, w, c$  $\text{Percept}([s, \text{None}, g, w, c], t) \Rightarrow \neg \text{Breeze}(t)$

∀ $t, s, b, w, c$  $\text{Percept}([s, b, \text{Glitter}, w, c], t) \Rightarrow \text{Glitter}(t)$

∀ $t, s, b, w, c$  $\text{Percept}([s, b, \text{None}, w, c], t) \Rightarrow \neg \text{Glitter}(t)$

⋮

∀ $t$  $\text{Glitter}(t) \Rightarrow \text{BestAction}(\text{Grab}, t)$ \hspace{1cm} // simple “reflex” behavior
Rules for the Environment

• Adjacency of two squares

square at row \( x \) and column \( y \)

\[ \forall x, y, a, b \quad \text{Adjacent}([x, y], [a, b]) \iff \\
(x = a \land (y = b - 1 \lor y = b + 1)) \lor \\
(y = b \land (x = a - 1 \lor x = a + 1)) \]

// if using proportional logic, we would have to name every square, say, \( \text{Square}_{i,j} \),
// for \( 1 \leq i, j \leq 4 \), and would need one such fact for 120 different pairs of squares!
Rules for the Environment

• Adjacency of two squares

square at row $x$ and column $y$

\[
\forall x, y, a, b \quad \text{Adjacent}([x, y], [a, b]) \iff \\
(x = a \land (y = b - 1 \lor y = b + 1)) \lor (y = b \land (x = a - 1 \lor x = a + 1))
\]

// if using proportional logic, we would have to name every square, say, Square$_{i,j}$,
// for $1 \leq i, j \leq 4$, and would need one such fact for 120 different pairs of squares!

• Unary predicate *Pit* (no reason to distinguish among pits).
Rules for the Environment

- **Adjacency of two squares**

  square at row $x$ and column $y$

  \[
  \forall x, y, a, b \quad \text{Adjacent}([x, y], [a, b]) \iff (x = a \land (y = b - 1 \lor y = b + 1)) \lor (y = b \land (x = a - 1 \lor x = a + 1))
  \]

  // if using proportional logic, we would have to name every square, say, Square$_{i,j}$, for $1 \leq i, j \leq 4$, and would need one such fact for 120 different pairs of squares!

- **Unary predicate $Pit$** (no reason to distinguish among pits).

  \[
  Pit([x, y]) \equiv true \quad \text{if and only if the square} \ [x, y] \ \text{contains a pit.}
  \]
Rules for the Environment

- Adjacency of two squares
  
  square at row $x$ and column $y$
  
  \[
  \forall x, y, a, b \quad \text{Adjacent}([x, y], [a, b]) \iff \\
  (x = a \land (y = b - 1 \lor y = b + 1)) \lor (y = b \land (x = a - 1 \lor x = a + 1))
  \]

  // if using proportional logic, we would have to name every square, say, Square$_{i,j}$, // for $1 \leq i, j \leq 4$, and would need one such fact for 120 different pairs of squares!

- Unary predicate $\text{Pit}$ (no reason to distinguish among pits).
  
  $\text{Pit}([x, y]) \equiv \text{true}$ if and only if the square $[x, y]$ contains a pit.

- Constants $\text{Agent}$, $\text{Wumpus}$
Rules for the Environment

- **Adjacency of two squares**

  square at row $x$ and column $y$

  $$\forall x, y, a, b \quad \text{Adjacent}(\{x, y\}, \{a, b\}) \iff (x = a \land (y = b - 1 \lor y = b + 1)) \lor (y = b \land (x = a - 1 \lor x = a + 1))$$

  // if using proportional logic, we would have to name every square, say, $\text{Square}_{i,j}$, // for $1 \leq i, j \leq 4$, and would need one such fact for 120 different pairs of squares!

- **Unary predicate Pit** (no reason to distinguish among pits).

  $$\text{Pit}(\{x, y\}) \equiv \text{true} \quad \text{if and only if the square } [x, y] \text{ contains a pit.}$$

- **Constants Agent, Wumpus**

- **Ternary predicate At to represent changing or non-changing locations**

  $$\forall t \quad \text{At}(\text{Wumpus}, [1, 3], t) \quad \text{// fixed location for the wumpus}$$
Rules for the Environment

• Adjacency of two squares

square at row $x$ and column $y$

$$\forall x, y, a, b \ \text{Adjacent}([x, y], [a, b]) \iff (x = a \land (y = b - 1 \lor y = b + 1)) \lor (y = b \land (x = a - 1 \lor x = a + 1))$$

// if using proportional logic, we would have to name every square, say, Square$_{i,j}$, // for $1 \leq i, j \leq 4$, and would need one such fact for 120 different pairs of squares!

• Unary predicate $Pit$ (no reason to distinguish among pits).

$$Pit([x, y]) \equiv \text{true} \quad \text{if and only if the square } [x, y] \text{ contains a pit.}$$

• Constants $Agent$, $Wumpus$

• Ternary predicate $At$ to represent changing or non-changing locations

$$\forall t \ At(Wumpus, [1, 3], t) \quad // \text{fixed location for the wumpus}$$

$$\forall x, s_1, s_2, t \ At(x,s_1, t) \land At(x,s_2, t) \Rightarrow s_1 = s_2 \quad // \text{only one location at a time}$$
Rules for Percepts, Actions and Inferences

- Percepts of breeze, stench, etc.

\[ \forall s, t \: \text{At}(\text{Agent}, s, t) \land \text{Breeze}(t) \Rightarrow \text{Breezy}(s) \]
Rules for Percepts, Actions and Inferences

- Percepts of breeze, stench, etc.

\[
\forall s, t \ At(Agent, s, t) \land Breeze(t) \Rightarrow Breezy(s)
\]

- Diagnostic (inferring cause from effect)

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

// if using proportional logic, we would need a separate axiom for every square.
Rules for Percepts, Actions and Inferences

• Percepts of breeze, stench, etc.

\[ \forall s, t \, \text{At}(\text{Agent}, s, t) \land \text{Breeze}(t) \Rightarrow \text{Breezy}(s) \]

• Diagnostic (inferring cause from effect)

\[ \forall s \, \text{Breezy}(s) \iff \exists r \, \text{Adjacent}(r, s) \land \text{Pit}(r) \]

// if using proportional logic, we would need a separate axiom for every square.

• Causal (inferring effect from cause)

\[ \forall s \, (\forall r \, \text{Adjacent}(r, s) \Rightarrow \neg \text{Pit}(r)) \Rightarrow \neg \text{Breezy}(s) \]
Rules for Percepts, Actions and Inferences

• Percepts of breeze, stench, etc.

\[ \forall s, t \ \text{At}(\text{Agent}, s, t) \land \text{Breeze}(t) \Rightarrow \text{Breezy}(s) \]

• Diagnostic (inferring cause from effect)

\[ \forall s \ \text{Breezy}(s) \iff \exists r \ \text{Adjacent}(r, s) \land \text{Pit}(r) \]

// if using proportional logic, we would need a separate axiom for every square.

• Causal (inferring effect from cause)

\[ \forall s \ (\forall r \ \text{Adjacent}(r, s) \Rightarrow \neg \text{Pit}(r)) \Rightarrow \neg \text{Breezy}(s) \]

• Quantification over time

\[ \forall t \ \text{HaveArrow}(t + 1) \iff \text{HaveArrow}(t) \land \neg \text{Action}(\text{Shoot}, t) \]
Rules for Percepts, Actions and Inferences

- Percepts of breeze, stench, etc.

\[ \forall s, t \; \text{At}(Agent, s, t) \land \text{Breeze}(t) \Rightarrow \text{Breezy}(s) \]

- **Diagnostic** (inferring cause from effect)

\[ \forall s \; \text{Breezy}(s) \Leftrightarrow \exists r \; \text{Adjacent}(r, s) \land \text{Pit}(r) \]

// if using proportional logic, we would need a separate axiom for every square.

- **Causal** (inferring effect from cause)

\[ \forall s \; (\forall r \; \text{Adjacent}(r, s) \Rightarrow \neg \text{Pit}(r)) \Rightarrow \neg \text{Breezy}(s) \]

- Quantification over time

\[ \forall t \; \text{HaveArrow}(t + 1) \Leftrightarrow \text{HaveArrow}(t) \land \neg \text{Action}(\text{Shoot}, t) \]

The first-order logic formulation is no less concise than the English description.
III. Knowledge Engineering Process

- Identification of questions and facts.

  What will the KB support and what facts are available?
III. Knowledge Engineering Process

- Identification of questions and facts.
  What will the KB support and what facts are available?

- Knowledge assembly or acquisition.
  Work with real experts to extract knowledge (not yet formally represented).
III. Knowledge Engineering Process

- Identification of questions and facts.
  What will the KB support and what facts are available?

- Knowledge assembly or acquisition.
  Work with real experts to extract knowledge (not yet formally represented).

- Decision on a vocabulary (predicates, functions, and constants).
  In the wumpus world, should pits be represented by objects or by a unary predicate on squares?
III. Knowledge Engineering Process

- Identification of questions and facts.
  What will the KB support and what facts are available?

- Knowledge assembly or acquisition.
  Work with real experts to extract knowledge (not yet formally represented).

- Decision on a vocabulary (predicates, functions, and constants).
  In the wumpus world, should pits be represented by objects or by a unary predicate on squares?

- Encoding of general knowledge (axioms).

- Formal description of the problem instance.
III. Knowledge Engineering Process

- Identification of questions and facts.
  What will the KB support and what facts are available?

- Knowledge assembly or acquisition.
  Work with real experts to extract knowledge (not yet formally represented).

- Decision on a vocabulary (predicates, functions, and constants).
  In the wumpus world, should pits be represented by objects or by a unary predicate on squares?

- Encoding of general knowledge (axioms).

- Formal description of the problem instance.

- Queries to and answers from the inference procedure.
  Derive the facts we are interested in knowing.
III. Knowledge Engineering Process

- **Identification of questions and facts.**
  
  What will the KB support and what facts are available?

- **Knowledge assembly or acquisition.**
  
  Work with real experts to extract knowledge (not yet formally represented).

- **Decision on a vocabulary (predicates, functions, and constants).**
  
  In the wumpus world, should pits be represented by objects or by a unary predicate on squares?

- **Encoding of general knowledge (axioms).**

- **Formal description of the problem instance.**

- **Queries to and answers from the inference procedure.**
  
  Derive the facts we are interested in knowing.

- **Debugging and evaluation of the KB.**
The Electronic Circuits Domain

- Four types of gates: AND, OR, XOR, NOT.
- 1 or 2 inputs, and 1 output
- Represent connections between terminals.

A one-bit full adder
The Electronic Circuits Domain

- Four types of gates: AND, OR, XOR, NOT.
- 1 or 2 inputs, and 1 output
- Represent connections between terminals.
- No need to represent wires or their paths.
- Component sizes, shapes, colors, or costs are irrelevant.

A one-bit full adder
The Electronic Circuits Domain

- Four types of gates: \textit{AND}, \textit{OR}, \textit{XOR}, \textit{NOT}.
  - 1 or 2 inputs, and 1 output
- Represent connections between terminals.
- No need to represent wires or their paths.
- Component sizes, shapes, colors, or costs are irrelevant.

Objects: actual gates and circuits. E.g., \(X_1, A_2, C_1\).
The Electronic Circuits Domain

- Four types of gates: \textit{AND}, \textit{OR}, \textit{XOR}, \textit{NOT}.
  - 1 or 2 inputs, and 1 output
  - Represent connections between terminals.
  - No need to represent wires or their paths.
  - Component sizes, shapes, colors, or costs are irrelevant.

- Objects: actual gates and circuits. E.g., \(X_1, A_2, C_1\).
- Predicates: \textit{Gate}, \textit{Terminal}, \textit{Circuit}
The Electronic Circuits Domain

- Four types of gates: AND, OR, XOR, NOT.
  - 1 or 2 inputs, and 1 output
- Represents connections between terminals.
- No need to represent wires or their paths.
- Component sizes, shapes, colors, or costs are irrelevant.

Objects: actual gates and circuits. E.g., $X_1$, $A_2$, $C_1$.

Predicates: Gate, Terminal, Circuit

Functions

- $Type(X_1)$: type of gate $X_1$ (which is XOR)
- $In(1, X_2)$: 1st input terminal for gate $X_2$
- $Out(2, C_1)$: 2nd output terminal for circuit $C_1$
- $Arity(A_1, 2, 1)$: two input terminals and one output terminal for the gate $A_1$
The Electronic Circuits Domain

- Four types of gates: **AND, OR, XOR, NOT**.
  - 1 or 2 inputs, and 1 output
- Represent connections between terminals.
- No need to represent wires or their paths.
- Component sizes, shapes, colors, or costs are irrelevant.

- Objects: actual gates and circuits. E.g., $X_1, A_2, C_1$.
- Predicates: **Gate, Terminal, Circuit**
- Functions
  - **Type($X_1$)**: type of gate $X_1$ (which is XOR)
  - **In(1, $X_2$)**: 1st input terminal for gate $X_2$
  - **Out(2, $C_1$)**: 2nd output terminal for circuit $C_1$
  - **Arity($A_1, 2, 1$)**: two input terminals and one output terminal for the gate $A_1$
- Connectivity predicate: **Connected(Out(1, $X_1$), In(1, $X_2$))**
The Electronic Circuits Domain

- Four types of gates: \textit{AND}, \textit{OR}, \textit{XOR}, \textit{NOT}.
  - 1 or 2 inputs, and 1 output
- Represent connections between terminals.
- No need to represent wires or their paths.
- Component sizes, shapes, colors, or costs are irrelevant.

- Objects: actual gates and circuits. E.g., \(X_1, A_2, C_1\).
- Predicates: \textit{Gate}, \textit{Terminal}, \textit{Circuit}
- Functions
  - \textit{Type}(X_1): type of gate \(X_1\) (which is \textit{XOR})
  - \textit{In}(1, X_2): 1\textsuperscript{st} input terminal for gate \(X_2\)
  - \textit{Out}(2, C_1): 2\textsuperscript{nd} output terminal for circuit \(C_1\)
  - \textit{Arity}(A_1, 2, 1): two input terminals and one output terminal for the gate \(A_1\)
- Connectivity predicate: \textit{Connected}(Out(1, X_1), In(1, X_2))
- Signal function: \textit{Signal}(t) has value 1 or 0 at time \(t\).
Encoding General Domain Knowledge

// Two connected terminals have the same signal.
\[
\forall t_1, t_2 \quad \text{Terminal}(t_1) \land \text{Terminal}(t_2) \land \text{Connected}(t_1, t_2) \Rightarrow \text{Signal}(t_1) = \text{Signal}(t_2)
\]
// Two connected terminals have the same signal.
∀t₁, t₂  \( Terminal(t₁) \land Terminal(t₂) \land Connected(t₁, t₂) \Rightarrow Signal(t₁) = Signal(t₂) \)

// Every terminal has signal that is either 1 or 0.
∀t  \( Terminal(t) \Rightarrow Signal(t) = 1 \lor Signal(t) = 0 \)
Encoding General Domain Knowledge

// Two connected terminals have the same signal.
\[ \forall t_1, t_2 \quad \text{Terminal}(t_1) \land \text{Terminal}(t_2) \land \text{Connected}(t_1, t_2) \Rightarrow \text{Signal}(t_1) = \text{Signal}(t_2) \]

// Every terminal has signal that is either 1 or 0.
\[ \forall t \quad \text{Terminal}(t) \Rightarrow \text{Signal}(t) = 1 \lor \text{Signal}(t) = 0 \]

// Connectivity is commutative.
\[ \forall t_1, t_2 \quad \text{Connected}(t_1, t_2) \iff \text{Connected}(t_2, t_1) \]
Encoding General Domain Knowledge

// Two connected terminals have the same signal.
\[ \forall t_1, t_2 \quad \text{Terminal}(t_1) \land \text{Terminal}(t_2) \land \text{Connected}(t_1, t_2) \Rightarrow \text{Signal}(t_1) = \text{Signal}(t_2) \]

// Every terminal has signal that is either 1 or 0.
\[ \forall t \quad \text{Terminal}(t) \Rightarrow \text{Signal}(t) = 1 \lor \text{Signal}(t) = 0 \]

// Connectivity is commutative.
\[ \forall t_1, t_2 \quad \text{Connected}(t_1, t_2) \iff \text{Connected}(t_2, t_1) \]

// Four types of gates.
\[ \forall g, k \quad \text{Gate}(g) \land k = \text{Type}(g) \Rightarrow k = \text{AND} \lor k = \text{OR} \lor k = \text{XOR} \lor k = \text{NOT} \]
Encoding General Domain Knowledge

// Two connected terminals have the same signal.
∀ t₁, t₂ Terminal(t₁) ∧ Terminal(t₂) ∧ Connected(t₁, t₂) ⇒ Signal(t₁) = Signal(t₂)

// Every terminal has signal that is either 1 or 0.
∀ t Terminal(t) ⇒ Signal(t) = 1 ∨ Signal(t) = 0

// Connectivity is commutative.
∀ t₁, t₂ Connected(t₁, t₂) ⇔ Connected(t₂, t₁)

// Four types of gates.
∀ g, k Gate(g) ∧ k = Type(g) ⇒ k = AND ∨ k = OR ∨ k = XOR ∨ k = NOT

// An AND gate outputs 0 if and only if any of its input is 0.
∀ g Gate(g) ∧ Type(g) = AND ⇒ (Signal(Out(1, g)) = 0 ⇔ ∃ n Signal(In(n, g)) = 0)
Encoding General Domain Knowledge

// Two connected terminals have the same signal.
∀ t₁, t₂  Terminal(t₁) ∧ Terminal(t₂) ∧ Connected(t₁, t₂) ⇒ Signal(t₁) = Signal(t₂)

// Every terminal has signal that is either 1 or 0.
∀ t  Terminal(t) ⇒ Signal(t) = 1 ∨ Signal(t) = 0

// Connectivity is commutative.
∀ t₁, t₂  Connected(t₁, t₂) ⇔ Connected(t₂, t₁)

// Four types of gates.
∀ g, k  Gate(g) ∧ k = Type(g) ⇒ k = AND ∨ k = OR ∨ k = XOR ∨ k = NOT

// An AND gate outputs 0 if and only if any of its input is 0.
∀ g  Gate(g) ∧ Type(g) = AND ⇒ (Signal(Out(1, g)) = 0 ⇔ ∃ n  Signal(In(n, g)) = 0)

// An OR gate outputs 1 if and only if any of its input is 1.
∀ g  Gate(g) ∧ Type(g) = OR ⇒ (Signal(Out(1, g)) = 1 ⇔ ∃ n  Signal(In(n, g)) = 1)
// An XOR gate outputs 1 if and only if its inputs are different.
\[ \forall g \quad \text{Gate}(g) \land \text{Type}(g) = \text{XOR} \Rightarrow (\text{Signal}(\text{Out}(1, g)) = 1 \iff \text{Signal}(\text{In}(1, g)) \neq \text{Signal}(\text{In}(2, g))) \]

// An NOT gate’s output is different from its input.
\[ \forall g \quad \text{Gate}(g) \land \text{Type}(g) = \text{NOT} \Rightarrow (\text{Signal}(\text{Out}(1, g)) = 1 \iff \text{Signal}(\text{Out}(1, g)) \neq \text{Signal}(\text{In}(1, g))) \]
// An XOR gate outputs 1 if and only if its inputs are different.
\[ \forall g \; \text{Gate}(g) \land \text{Type}(g) = \text{XOR} \Rightarrow (\text{Signal}(\text{Out}(1, g)) = 1 \iff \text{Signal}(\text{In}(1, g)) \neq \text{Signal}(\text{In}(2, g))) \]

// An NOT gate’s output is different from its input.
\[ \forall g \; \text{Gate}(g) \land \text{Type}(g) = \text{NOT} \Rightarrow (\text{Signal}(\text{Out}(1, g)) = 1 \iff \text{Signal}(\text{Out}(1, g)) \neq \text{Signal}(\text{In}(1, g))) \]

// All the gates (except for NOT) have two inputs and one output.
\[ \forall g \; \text{Gate}(g) \land \text{Type}(g) = \text{NOT} \Rightarrow \text{Arity}(g, 1, 1) \]
\[ \forall g, k \; \text{Gate}(g) \land k = \text{Type}(g) \land (k = \text{AND} \lor k = \text{OR} \lor k = \text{XOR}) \Rightarrow \text{Arity}(g, 2, 1) \]
// An XOR gate outputs 1 if and only if its inputs are different.
\[ \forall g \ Gate(g) \land Type(g) = XOR \Rightarrow (Signal(Out(1, g))) = 1 \iff Signal(In(1, g)) \neq Signal(In(2, g)) \]

// An NOT gate’s output is different from its input.
\[ \forall g \ Gate(g) \land Type(g) = NOT \Rightarrow (Signal(Out(1, g))) = 1 \iff Signal(Out(1, g)) \neq Signal(In(1, g)) \]

// All the gates (except for NOT) have two inputs and one output.
\[ \forall g \ Gate(g) \land Type(g) = NOT \Rightarrow \text{Arity}(g, 1, 1) \]
\[ \forall g, k \ Gate(g) \land k = \text{Type}(g) \land (k = \text{AND} \lor k = \text{OR} \lor k = \text{XOR}) \Rightarrow \text{Arity}(g, 2, 1) \]

// A circuit has terminals exactly up to its input and output arity.
\[ \forall c, i, j \ Circuit(c) \land \text{Arity}(c, i, j) \Rightarrow \\
\forall n \ (n \leq i \Rightarrow \text{Terminal}(In(n, c)) \land (n > i \Rightarrow \text{In}(n, c) = \text{Nothing})) \land \\
\forall n \ (n \leq j \Rightarrow \text{Terminal}(Out(n, c)) \land (n > j \Rightarrow \text{Out}(n, c) = \text{Nothing})) \]
// An XOR gate outputs 1 if and only if its inputs are different.
\[ \forall g \ Gate(g) \land Type(g) = XOR \Rightarrow (Signal(Out(1, g)) = 1 \iff Signal(In(1, g)) \neq Signal(In(2, g))) \]

// An NOT gate’s output is different from its input.
\[ \forall g \ Gate(g) \land Type(g) = NOT \Rightarrow \left( Signal(Out(1, g)) = 1 \iff Signal(Out(1, g)) \neq Signal(In(1, g)) \right) \]

// All the gates (except for NOT) have two inputs and one output.
\[ \forall g \ Gate(g) \land Type(g) = NOT \Rightarrow Arity(g, 1, 1) \]
\[ \forall g, k \ Gate(g) \land k = Type(g) \land (k = AND \lor k = OR \lor k = XOR) \Rightarrow Arity(g, 2, 1) \]

// A circuit has terminals exactly up to its input and output arity.
\[ \forall c, i, j \ Circuit(c) \land Arity(c, i, j) \Rightarrow \]
\[ \forall n \ (n \leq i \Rightarrow Terminal(In(n, c)) \land (n > i \Rightarrow In(n, c) = Nothing)) \land \]
\[ \forall n \ (n \leq j \Rightarrow Terminal(Out(n, c)) \land (n > j \Rightarrow Out(n, c) = Nothing)) \]

// Gates and terminals are all distinct.
\[ \forall g, t, s \ Gate(g) \land Terminal(t) \land Signal(s) \Rightarrow g \neq t \land g \neq s \land t \neq s \]
// An XOR gate outputs 1 if and only if its inputs are different.
∀ \text{Gate}(g) \land \text{Type}(g) = \text{XOR} \Rightarrow \left( \text{Signal(Out}(1,g)) = 1 \iff \text{Signal(In}(1,g)) \neq \text{Signal(In}(2,g)) \right)

// An NOT gate’s output is different from its input.
∀ \text{Gate}(g) \land \text{Type}(g) = \text{NOT} \Rightarrow \left( \text{Signal(Out}(1,g)) = 1 \iff \text{Signal(Out}(1,g)) \neq \text{Signal(In}(1,g)) \right)

// All the gates (except for NOT) have two inputs and one output.
∀ \text{Gate}(g) \land \text{Type}(g) = \text{NOT} \Rightarrow \text{Arity}(g,1,1)
∀ \text{Gate}(g) \land k=\text{Type}(g) \land (k = \text{AND} \lor k = \text{OR} \lor k = \text{XOR}) \Rightarrow \text{Arity}(g,2,1)

// A circuit has terminals exactly up to its input and output arity.
∀ \text{Circuit}(c) \land \text{Arity}(c,i,j) \Rightarrow
∀ n \ (n \leq i \Rightarrow \text{Terminal(In}(n,c)) \land (n > i \Rightarrow \text{In}(n,c) = \text{Nothing})) \land
∀ n \ (n \leq j \Rightarrow \text{Terminal(Out}(n,c)) \land (n > j \Rightarrow \text{Out}(n,c) = \text{Nothing}))

// Gates and terminals are all distinct.
∀ \text{Gate}(g) \land \text{Terminal}(t) \land \text{Signal}(s) \Rightarrow g \neq t \land g \neq s \land t \neq s
// An XOR gate outputs 1 if and only if its inputs are different.
∀g \ Gate(g) \land \ Type(g) = \text{XOR} \Rightarrow (\text{Signal(Out}(1, g)) = 1 \iff \text{Signal(In}(1, g)) \neq \text{Signal(In}(2, g)))

// An NOT gate's output is different from its input.
∀g \ Gate(g) \land \ Type(g) = \text{NOT} \Rightarrow (\text{Signal(Out}(1, g)) = 1 \iff \text{Signal(Out}(1, g)) \neq \text{Signal(In}(1, g)))

// All the gates (except for NOT) have two inputs and one output.
∀g \ Gate(g) \land \ Type(g) = \text{NOT} \Rightarrow \text{Arity}(g, 1, 1)
∀g, k \ Gate(g) \land k=\text{Type}(g) \land (k = \text{AND} \lor k = \text{OR} \lor k = \text{XOR}) \Rightarrow \text{Arity}(g, 2, 1)

// A circuit has terminals exactly up to its input and output arity.
∀c, i, j \ Circuit(c) \land \text{Arity}(c, i, j) \Rightarrow
∀n (n \leq i \Rightarrow \text{Terminal(In}(n, c)) \land (n > i \Rightarrow \text{In}(n, c) = \text{Nothing})) \land
∀n (n \leq j \Rightarrow \text{Terminal(Out}(n, c)) \land (n > j \Rightarrow \text{Out}(n, c) = \text{Nothing}))

// Gates and terminals are all distinct.
∀g, t, s \ Gate(g) \land \text{Terminal(t)} \land \text{Signal(s)} \Rightarrow g \neq t \land g \neq s \land t \neq s

// Gates are circuits.
∀g \ Gate(g) \Rightarrow \text{Circuit}(g)
Encoding a Problem Instance

Circuit and component gates:

\[
\begin{align*}
\text{Circuit}(C_1) & \wedge \text{Arity}(C_1, 3, 2) \\
\text{Gate}(X_1) & \wedge \text{Type}(X_1) = \text{XOR} \\
\text{Gate}(X_2) & \wedge \text{Type}(X_2) = \text{XOR} \\
\text{Gate}(A_1) & \wedge \text{Type}(A_1) = \text{AND} \\
\text{Gate}(A_2) & \wedge \text{Type}(A_2) = \text{AND} \\
\text{Gate}(O_1) & \wedge \text{Type}(O_1) = \text{OR}
\end{align*}
\]
Encoding a Problem Instance

Circuit and component gates:

- \( \text{Circuit}(C_1) \land \text{Arity}(C_1, 3, 2) \)
- \( \text{Gate}(X_1) \land \text{Type}(X_1) = \text{XOR} \)
- \( \text{Gate}(X_2) \land \text{Type}(X_2) = \text{XOR} \)
- \( \text{Gate}(A_1) \land \text{Type}(A_1) = \text{AND} \)
- \( \text{Gate}(A_2) \land \text{Type}(A_2) = \text{AND} \)
- \( \text{Gate}(O_1) \land \text{Type}(O_1) = \text{OR} \)

Connections between the circuit and component gates:

- \( \text{Connected}(\text{Out}(1, X_1), \text{In}(1, X_2)) \)
- \( \text{Connected}(\text{Out}(1, X_1), \text{In}(2, A_2)) \)
- \( \text{Connected}(\text{Out}(1, A_2), \text{In}(1, O_1)) \)
- \( \text{Connected}(\text{Out}(1, A_1), \text{In}(2, O_1)) \)
- \( \text{Connected}(\text{Out}(1, X_2), \text{Out}(1, C_1)) \)
- \( \text{Connected}(\text{Out}(1, O_1), \text{Out}(2, C_1)) \)
- \( \text{Connected}(\text{In}(1, C_1), \text{In}(1, X_1)) \)
- \( \text{Connected}(\text{In}(1, C_1), \text{In}(1, A_1)) \)
- \( \text{Connected}(\text{In}(2, C_1), \text{In}(2, X_1)) \)
- \( \text{Connected}(\text{In}(2, C_1), \text{In}(2, A_1)) \)
- \( \text{Connected}(\text{In}(3, C_1), \text{In}(2, X_2)) \)
- \( \text{Connected}(\text{In}(3, C_1), \text{In}(1, A_2)) \)
Q. What combinations of inputs would cause the first output of $C_1$ to be 0 and its second output to be 1?
Q. What combinations of inputs would cause the first output of \( C_1 \) to be 0 and its second output to be 1?

\[
\exists i_1, i_2, i_3 \ \text{Signal}(\text{In}(1, C_1)) = i_1 \land \text{Signal}(\text{In}(2, C_1)) = i_2 \land \text{Signal}(\text{In}(3, C_1)) = i_3 \\
\land \text{Signal}(\text{Out}(1, C_1)) = 0 \land \text{Signal}(\text{Out}(2, C_1)) = 1
\]
Q. What combinations of inputs would cause the first output of $C_1$ to be 0 and its second output to be 1?

$$\exists i_1, i_2, i_3 \ Signal(In(1, C_1)) = i_1 \land Signal(In(2, C_1)) = i_2 \land Signal(In(3, C_1)) = i_3 \land Signal(Out(1, C_1)) = 0 \land Signal(Out(2, C_1)) = 1$$

ASKVARS will give three substitutions as answers.

$$\{i_1/1, i_2/1, i_3/0\} \quad \{i_1/1, i_2/0, i_3/1\} \quad \{i_1/0, i_2/1, i_3/1\}$$
Queries

Q. What combinations of inputs would cause the first output of $C_1$ to be 0 and its second output to be 1?

$$\exists i_1, i_2, i_3 \ Signal(In(1, C_1)) = i_1 \land Signal(In(2, C_1)) = i_2 \land Signal(In(3, C_1)) = i_3 \land Signal(Out(1, C_1)) = 0 \land Signal(Out(2, C_1)) = 1$$

ASK\VARS will give three substitutions as answers.

$$\{i_1/1, i_2/1, i_3/0\} \quad \{i_1/1, i_2/0, i_3/1\} \quad \{i_1/0, i_2/1, i_3/1\}$$

Debugging: We can also perturb the KB to see what erroneous behaviors would emerge, and then identify missing rules for instance.
IV. Propositional vs. First-Order Inference

One way of inference is to convert the first-order KB to propositional logic.
One way of inference is to convert the first-order KB to propositional logic.

- Eliminate universal quantifiers.

\[ \forall x \ Human(x) \Rightarrow Fallible(x) \]  // All humans are fallible.
IV. Propositional vs. First-Order Inference

One way of inference is to convert the first-order KB to propositional logic.

- Eliminate universal quantifiers.

\[ \forall x \ Human(x) \Rightarrow Fallible(x) \]  // All humans are fallible.

We can infer sentences like

\[
\begin{align*}
\text{Human}(\text{Socrates}) & \Rightarrow \text{Fallible}(\text{Socrates}) \\
\text{Human}(\text{Einstein}) & \Rightarrow \text{Fallible}(\text{Einstein}) \\
\text{Human}(\text{Messi}) & \Rightarrow \text{Fallible}(\text{Messi}) \\
\vdots
\end{align*}
\]
IV. Propositional vs. First-Order Inference

One way of inference is to convert the first-order KB to propositional logic.

- Eliminate universal quantifiers.

\[ \forall x \; \text{Human}(x) \Rightarrow \text{Fallible}(x) \] // All humans are fallible.

We can infer sentences like

\[
\begin{align*}
\text{Human}(\text{Socrates}) & \Rightarrow \text{Fallible}(\text{Socrates}) \\
\text{Human}(\text{Einstein}) & \Rightarrow \text{Fallible}(\text{Einstein}) \\
\text{Human}(\text{Messi}) & \Rightarrow \text{Fallible}(\text{Messi}) \\
\vdots \\
\forall x \; \text{Bird}(x) & \Rightarrow \text{WarmBlooded}(x) \land \text{HaveWings}(x)
\end{align*}
\] // All birds are warm-blooded and have wings.
IV. Propositional vs. First-Order Inference

One way of inference is to convert the first-order KB to propositional logic.

- Eliminate universal quantifiers.

\[ \forall x \; \text{Human}(x) \Rightarrow \text{Fallible}(x) \]  
// All humans are fallible.

We can infer sentences like

\[ \text{Human}(\text{Socrates}) \Rightarrow \text{Fallible}(\text{Socrates}) \]
\[ \text{Human}(\text{Einstein}) \Rightarrow \text{Fallible}(\text{Einstein}) \]
\[ \text{Human}(\text{Messi}) \Rightarrow \text{Fallible}(\text{Messi}) \]
\[ \vdots \]

From

// All birds are warm-blooded and have wings.
\[ \forall x \; \text{Bird}(x) \Rightarrow \text{WarmBlooded}(x) \land \text{HaveWings}(x) \]

We can infer (if \text{Bird}(\text{Ostrich}) and \text{Bird}(\text{Peacock}) are in the KB):

\[ \text{WarmBlooded}(\text{Ostrich}) \land \text{HaveWings}(\text{Ostrich}) \]
\[ \text{WarmBlooded}(\text{Peacock}) \land \text{HaveWings}(\text{Peacock}) \]
Universal and Existential Instantiations

A ground term in FOL is a term without variables.
Universal and Existential Instantiations

A *ground term* in FOL is a term without variables.

Substitute a ground term for a universally quantified variable.

\[ \forall v \alpha \]

\[ \text{SUBST}\{v/g\}, \alpha \]
A ground term in FOL is a term without variables.

Substitute a ground term for a universally quantified variable.

\[
\text{sentence} \\
\forall v \alpha \\
\text{SUBST}\{{v/g}, \alpha\}
\]
Universal and Existential Instantiations

A *ground term* in FOL is a term without variables.

Substitute a ground term for a universally quantified variable.

\[ \forall v \alpha \]

\[ \text{SUBST} \left( \{ v/g \}, \alpha \right) \]

sentence

substitution \( \theta \), e.g., \( \theta = \{ x/Socrates \} \)
A *ground term* in FOL is a term without variables.

Substitute a ground term for a universally quantified variable.

\[ \forall v \alpha \]

\[ \text{SUBST} \left( \{ v/g \}, \alpha \right) \]

sentence

substitution \( \theta \), e.g., \( \theta = \{ x/Socrates \} \)

E.g., \( \theta = \{ x/Ostrich \} \)

\[ \forall x \text{ Bird}(x) \Rightarrow \text{WarmBlooded}(x) \land \text{HaveWings}(x) \]
A *ground term* in FOL is a term without variables.

Substitute a ground term for a universally quantified variable.

\[ \forall v \alpha \]

\[ \text{SUBST}(\{v/g\}, \alpha) \]

substitution \( \theta \), e.g., \( \theta = \{x/Socrates\} \)

E.g., \( \theta = \{x/Ostrich\} \)

\[ \forall x \text{ Bird}(x) \Rightarrow \text{WarmBlooded}(x) \land \text{HaveWings}(x) \]
Universal and Existential Instantiations

A *ground term* in FOL is a term without variables.

Substitute a ground term for a universally quantified variable.

\[ \forall \alpha \]

\[ \text{SUBST}(\{v/g\}, \alpha) \]

sentence

substitution \( \theta \), e.g., \( \theta = \{x/Socrates\} \)

E.g., \( \theta = \{x/Ostrich\} \)

\[ \forall x \text{ Bird}(x) \Rightarrow \text{WarmBlooded}(x) \land \text{HaveWings}(x) \]

\[ \text{SUBST}(\theta, \alpha) \equiv \text{Bird}(Ostrich) \Rightarrow \text{WarmBlooded}(Ostrich) \land \text{HaveWings}(Ostrich) \]
Existential Instantiation

Substitute a single new constant symbol for an existentially quantified variable.

\[
\exists v \alpha
\]

\[
\text{SUBST}\{v/g\}, \alpha
\]
Existential Instantiation

Substitute a single new constant symbol for an existentially quantified variable.

From

\[ \exists y \text{ Mother}(y, \text{Liam}) \]
Existential Instantiation

Substitute a single new constant symbol for an existentially quantified variable.

\[
\exists v \alpha \\
\text{SUBST}\{(v/g), \alpha\}
\]

From

\[
\exists y \text{Mother}(y, Liam)
\]

we can infer

\[
\text{Mother}(\text{LiamsMom}, Liam)
\]

as long as \text{LiamsMom} does not appear elsewhere in the KB.
Existential Instantiation

Substitute a single new constant symbol for an existentially quantified variable.

\[ \exists v \alpha \]

\[ \text{SUBST}(\{v/g\}, \alpha) \]

From

\[ \exists y \text{Mother}(y, Liam) \]

we can infer

\[ \text{Mother}(\text{LiamsMom}, Liam) \]

as long as \text{LiamsMom} does not appear elsewhere in the KB.

\[ \text{Skolem constant} \]
Propositionalization (Non-Standard Way)

\[ \forall x \forall y \quad \text{Ancestor}(x, y) \rightarrow \text{Parent}(x, y) \lor \exists z (\text{Ancestor}(x, z) \land \text{Ancestor}(z, y)) \]

KB:

- Ancestor(John, David)
- Parent(John, David)
- Parent(David, Lisa)
Propositionalization (Non-Standard Way)

\[ \forall x \forall y \ Ancestor(x, y) \rightarrow Parent(x, y) \lor \exists z (Ancestor(x, z) \land Ancestor(z, y)) \]

**KB:**
- Ancestor(John, David)
- Parent(John, David)
- Parent(David, Lisa)

Ancestor(John, David)
\[ \rightarrow Parent(John, David) \lor (Ancestor(John, JohnDavidAnc) \land Ancestor(JohnDavidAnc, David)) \]
Propositionalization (Non-Standard Way)

$$\forall x \forall y \ Ancestor(x, y) \rightarrow Parent(x, y) \lor \exists z (Ancestor(x, z) \land Ancestor(z, y))$$

**KB:**

- Ancestor(John, David)
- Parent(John, David)
- Parent(David, Lisa)

Ancestor(John, David)
$$\rightarrow Parent(John, David) \lor (Ancestor(John, JohnDavidAnc) \land Ancestor(JohnDavidAnc, David))$$

Ancestor(David, John)
$$\rightarrow Parent(David, John) \lor (Ancestor(David, DavidJohnAnc) \land Ancestor(DavidJohnAnc, John))$$

Ancestor(John, Lisa)
$$\rightarrow Parent(John, Lisa) \lor (Ancestor(John, JohnLisaAnc) \land Ancestor(JohnLisaAnc, Lisa))$$

...
Skolemization (Standard Way)

A more standard way to eliminate an existential quantifier is to introduce a new function symbol, which is, however, not applicable in generating a PL sentence.

$$\exists y \ P(y, x_1, \ldots, x_n)$$
Skolemization (Standard Way)

A more standard way to eliminate an existential quantifier is to introduce a new function symbol, which is, however, not applicable in generating a PL sentence.

\[ \exists y \ P(y, x_1, \ldots, x_n) \quad \text{// } y \text{ depends on } x_1, \ldots, x_n \]
A more standard way to eliminate an existential quantifier is to introduce a new function symbol, which is, however, not applicable in generating a PL sentence.

\[ \exists y \ P(y, x_1, \ldots, x_n) \quad \text{// } y \text{ depends on } x_1, \ldots, x_n \]

eliminate \( y \) by introducing function \( f \)

\[ P(f(x_1, \ldots, x_n), x_1, \ldots, x_n) \]
Skolemization (Standard Way)

A more standard way to eliminate an existential quantifier is to introduce a new function symbol, which is, however, not applicable in generating a PL sentence.

\[
\exists y \ P(y, x_1, \ldots, x_n) \quad \text{// } y \text{ depends on } x_1, \ldots, x_n
\]

eliminate \( y \) by introducing function \( f \)

\[
P(f(x_1, \ldots, x_n), x_1, \ldots, x_n)
\]

- That \( P(y, x_1, \ldots, x_n) = \text{true} \) implicitly defines \( y \) as a function of \( x_1, \ldots, x_n \) (analogous to the implicit function theorem in multivariate calculus).
A more standard way to eliminate an existential quantifier is to introduce a new function symbol, which is, however, not applicable in generating a PL sentence.

\[ \exists y \ P(y, x_1, \ldots, x_n) \quad /\quad y \text{ depends on } x_1, \ldots, x_n \]

eliminate \( y \) by introducing function \( f \)

\[ P(f(x_1, \ldots, x_n), x_1, \ldots, x_n) \]

That \( P(y, x_1, \ldots, x_n) = \text{true} \) implicitly defines \( y \) as a function of \( x_1, \ldots, x_n \) (analogous to the implicit function theorem in multivariate calculus).

\[ \exists y \ Mother(y, Liam) \]

\[ \exists y \ Mother(y, Sophia) \]
Skolemization (Standard Way)

A more standard way to eliminate an existential quantifier is to introduce a new function symbol, which is, however, not applicable in generating a PL sentence.

\[ \exists y \ P(y, x_1, \ldots, x_n) \quad \text{// } y \text{ depends on } x_1, \ldots, x_n \]

eliminate \( y \) by introducing function \( f \)

\[ P(f(x_1, \ldots, x_n), x_1, \ldots, x_n) \]

That \( P(y, x_1, \ldots, x_n) = \text{true} \) implicitly defines \( y \) as a function of \( x_1, \ldots, x_n \) (analogous to the implicit function theorem in multivariate calculus).

\[ \exists y \ Mother(y, Liam) \]

new unary function \( \text{mom()} \)

\[ Mother(\text{mom}(Liam), Liam) \]

\[ \exists y \ Mother(y, Sophia) \]

\[ Mother(\text{mom}(Sophia), Sophia) \]
Skolemization (Standard Way)

A more standard way to eliminate an existential quantifier is to introduce a **new function symbol**, which is, however, not applicable in generating a PL sentence.

\[ \exists y \ P(y, x_1, \ldots, x_n) \quad \text{\tiny \small // } y \text{ depends on } x_1, \ldots, x_n \]

\[ \downarrow \quad \text{eliminate } y \text{ by introducing} \]

\[ f(x_1, \ldots, x_n), x_1, \ldots, x_n \]

\[ P(f(x_1, \ldots, x_n), x_1, \ldots, x_n) \]

- That \( P(y, x_1, \ldots, x_n) = \text{true} \) implicitly defines \( y \) as a function of \( x_1, \ldots, x_n \) (analogous to the implicit function theorem in multivariate calculus).

\[ \exists y \ \text{Mother}(y, Liam) \quad \text{new unary function } \text{mom}() \]

\[ \text{Mother(mom(Liam), Liam)} \]

\[ \exists y \ \text{Mother}(y, Sophia) \]

\[ \text{Mother(mom(Sophia), Sophia)} \]

- **Advantage**: one function instead of two new constants to denote the moms of Liam and Sophia.
Generalized Modus Ponens

\[(p_1 \land p_2 \land \cdots \land p_n \Rightarrow q), \quad p'_1, p'_2, \ldots, p'_n\]

Suppose there exists a substitution \( \theta \) such that

\[
\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p'_i) \quad \text{for } 1 \leq i \leq n
\]

Example \( n = 1, \ p_1 \equiv \text{Dad}(x, \text{John}), \ p'_1 \equiv \text{Dad}(\text{David}, y), \ \theta = \{x/\text{David}, \ y/\text{John}\} \)
**Generalized Modus Ponens**

\[(p_1 \land p_2 \land \cdots \land p_n \Rightarrow q), \quad p'_1, p'_2, \ldots, p'_n\]

Suppose there exists a substitution \(\theta\) such that

\[
\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p'_i) \quad \text{for} \quad 1 \leq i \leq n
\]

**Example** \(n = 1, \ p_1 \equiv \text{Dad}(x, John), \ p'_1 \equiv \text{Dad}(David, y), \ \theta = \{x/\text{David}, \ y/\text{John}\}\)

Then

\[
(p'_1, p'_2, \ldots, p'_n, \quad (p_1 \land p_2 \land \cdots \land p_n \Rightarrow q))
\]

\[
\text{SUBST}(\theta, q)
\]
Generalized Modus Ponens

\[(p_1 \land p_2 \land \cdots \land p_n \Rightarrow q), \quad p_1', p_2', \ldots, p_n'
\]

Suppose there exists a substitution \( \theta \) such that

\[
\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i') \quad \text{for} \quad 1 \leq i \leq n
\]

Example \( n = 1, p_1 \equiv \text{Dad}(x, \text{John}), p_1' \equiv \text{Dad}(\text{David}, y), \theta = \{x/\text{David}, \ y/\text{John}\} \)

Then

\[
p_1', p_2', \ldots, p_n', \quad (p_1 \land p_2 \land \cdots \land p_n \Rightarrow q)
\]

\[
\text{SUBST}(\theta, q)
\]

KB:

\[
\text{Gate}(X_1), \quad \text{Terminal}(\text{In}(1, C_1)) \quad \text{Gate}(g) \land \text{Terminal}(t) \Rightarrow g \neq t
\]
Generalized Modus Ponens

\[(p_1 \land p_2 \land \cdots \land p_n \Rightarrow q), \quad p'_1, p'_2, \ldots, p'_n\]

Suppose there exists a substitution \(\theta\) such that

\[\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p'_i) \quad \text{for } 1 \leq i \leq n\]

Example \(n = 1, p_1 \equiv \text{Dad}(x, \text{John}), p'_1 \equiv \text{Dad}(\text{David}, y), \theta = \{x/\text{David}, y/\text{John}\}\)

Then

\[p'_1, p'_2, \ldots, p'_n, \quad (p_1 \land p_2 \land \cdots \land p_n \Rightarrow q)\]

\[\text{SUBST}(\theta, q)\]

KB:

\[\text{Gate}(X_1), \quad \text{Terminal}(\text{In}(1, C_1))\]

\[\text{Gate}(g) \land \text{Terminal}(t) \Rightarrow g \neq t\]

\[\theta = \{g/X_1, t/(\text{In}(1, C_1))\}\]

\[q \text{ is } g \neq t\]
Generalized Modus Ponens

\((p_1 \land p_2 \land \cdots \land p_n \Rightarrow q), \ p'_1, p'_2, ..., p'_n\)

Suppose there exists a substitution \(\theta\) such that

\[\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p'_i) \quad \text{for } 1 \leq i \leq n\]

Example \(n = 1, \ p_1 \equiv \text{Dad}(x, \text{John}), \ p'_1 \equiv \text{Dad}(\text{David}, y), \ \theta = \{x/\text{David}, \ y/\text{John}\}\)

Then

\[p'_1, p'_2, ..., p'_n, \quad (p_1 \land p_2 \land \cdots \land p_n \Rightarrow q)\]

\[\text{SUBST}(\theta, q)\]

KB:

\[\text{Gate}(X_1), \ \text{Terminal}(\text{In}(1, C_1))\]

\[\text{Gate}(g) \land \text{Terminal}(t) \Rightarrow g \neq t\]

\[\text{SUBST}(\theta, q) \downarrow \theta = \{g/X_1,t/(\text{In}(1, C_1))\}\]

\(q\) is \(g \neq t\)

\[X_1 \neq \text{In}(1, C_1)\]
V. Unification

- The process of finding substitutions that make different logical expressions look identical.

- Carried out by the algorithm UNIFY.

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

- The process of finding substitutions that make different logical expressions look identical.
- Carried out by the algorithm UNIFY.

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

- The process of finding substitutions that make different logical expressions look identical.

- Carried out by the algorithm UNIFY.

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

Query: \text{AskVars(Knows(John, x))} \quad // \text{what does John know?}
V. Unification

- The process of finding substitutions that make different logical expressions look identical.
- Carried out by the algorithm UNIFY.

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

Query: \( \text{AskVars}(\text{Knows}(\text{John}, x)) \) // what does John know?

Answers: all the sentences in the KB found to unify with \text{Knows}(\text{John}, x).
V. Unification

- The process of finding substitutions that make different logical expressions look identical.

- Carried out by the algorithm UNIFY.

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

**Query:** \[ \text{AskVars}(\text{Knows}(John, x)) \] // what does John know?

**Answers:** all the sentences in the KB found to unify with \( \text{Knows}(John, x) \).

\[ \text{UNIFY}(\text{Knows}(John, x), \text{Knows}(John, Jane)) = \{x/\text{Jane}\} \]

\[ \text{UNIFY}(\text{Knows}(John, x), \text{Knows}(y, Bill)) = \{x/\text{Bill}, y/John\} \]

\[ \text{UNIFY}(\text{Knows}(John, x), \text{Knows}(y, \text{Mother}(y))) = \{y/John, x/\text{Mother}(John)\} \]
Unification (cont’d)

- Conflicting substitutions

\[ \text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) = \text{failure} \]
Unification (cont’d)

- Conflicting substitutions

\[
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) = \text{failure}
\]

\{x/\text{John}\}
Unification (cont’d)

[min][1] Conflicting substitutions

\[
\text{UNIFY}(\text{Knows}(John, x), \text{Knows}(x, Elizabeth)) = \text{failure}
\]

\[
\begin{align*}
\{x/John\} & \quad \{x/Elizabeth\}
\end{align*}
\]
Unification (cont’d)

- Conflicting substitutions

\[
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) = \text{failure}
\]

\{x/\text{John}\} \quad \{x/\text{Elizabeth}\}

x cannot take on the values John and Elizabeth at the same time!
Unification (cont’d)

- Conflicting substitutions

\[
\text{UNIFY(\text{Knows(John, } x\text{), Knows}(x, \text{Elizabeth}))} = \text{failure}
\]

\[
\{x/\text{John}\} \quad \{x/\text{Elizabeth}\}
\]

\(x\) cannot take on the values \text{John} and \text{Elizabeth} at the same time!

- Multiple unifiers

\[
\text{UNIFY(\text{Knows(John, } x\text{), Knows}(y, z))}
\]

could return \(\{y/\text{John, } x/z\}\)

or \(\{y/\text{John, } x/\text{John, } z/\text{John}\}\)
Unification (cont’d)

✧ Conflicting substitutions

\[
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) = \text{failure}
\]

\[
\{x/\text{John}\} \quad \{x/\text{Elizabeth}\}
\]

\(x\) cannot take on the values \(\text{John}\) and \(\text{Elizabeth}\) at the same time!

✧ Multiple unifiers

\[
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, z))
\]

could return \(\{y/\text{John}, x/z\}\)

or \(\{y/\text{John}, x/\text{John}, z/\text{John}\}\)

\(\rightarrow\) \(\text{Knows}(\text{John}, z)\)
Unification (cont’d)

♦ Conflicting substitutions

\[
\text{\textsc{unify}}(\text{Knows(John, } x\text{)}, \text{Knows}(x, \text{Elizabeth})) = \text{failure}
\]

\[
\{x/\text{John}\} \quad \{x/\text{Elizabeth}\}
\]

\(x\) cannot take on the values \text{John} and \text{Elizabeth} at the same time!

♦ Multiple unifiers

\[
\text{\textsc{unify}}(\text{Knows(John, } x\text{)}, \text{Knows}(y, z))
\]

could return \(\{y/\text{John}, x/z\}\) \(\rightarrow\) \text{Knows(John, } z\text{)}

or \(\{y/\text{John, } x/\text{John, } z/\text{John}\}\) \(\rightarrow\) \text{Knows(John, John)}
Unification (cont’d)

- Conflicting substitutions

\[
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) = \text{failure}
\]

\[
\{x/\text{John}\} \quad \{x/\text{Elizabeth}\}
\]

- Multiple unifiers

\[
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, z))
\]

could return \( \{y/\text{John}, x/z\} \quad \text{\rightarrow} \quad \text{Knows}(\text{John}, z) \)

more general unifier for fewer restriction on variable values

or \( \{y/\text{John}, x/\text{John}, z/\text{John}\} \quad \text{\rightarrow} \quad \text{Knows}(\text{John}, \text{John}) \)

\( x \) cannot take on the values \( \text{John} \) and \( \text{Elizabeth} \) at the same time!
Unification (cont’d)

- Conflicting substitutions

\[
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) = \text{failure}
\]

\[
\{x/\text{John}\} \quad \{x/\text{Elizabeth}\}
\]

\(x\) cannot take on the values \text{John} and \text{Elizabeth} at the same time!

- Multiple unifiers

\[
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, z))
\]

could return \(\{y/\text{John}, x/z\}\)  
more general unifier for fewer restriction on variable values

or \(\{y/\text{John}, x/\text{John}, z/\text{John}\}\)  
less general unifier
Unification (cont’d)

♠ Conflicting substitutions

\[
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) = \text{failure}
\]

\[
\{x/\text{John}\} \quad \{x/\text{Elizabeth}\}
\]

\(x\) cannot take on the values \textit{John} and \textit{Elizabeth} at the same time!

♠ Multiple unifiers

\[
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, z))
\]

could return \(\{y/\text{John}, x/z\}\) more general unifier for fewer restriction on variable values

or \(\{y/\text{John}, x/\text{John}, z/\text{John}\}\) less general unifier

\[
\text{Unification}(\text{John}, \text{John})
\]
Unification Algorithm

function UNIFY($x$, $y$, $\theta=$empty) returns a substitution to make $x$ and $y$ identical, or failure
  if $\theta =$ failure then return failure
  else if $x = y$ then return $\theta$
  else if VARIABLE?($x$) then return UNIFY-VAR($x$, $y$, $\theta$)
  else if VARIABLE?($y$) then return UNIFY-VAR($y$, $x$, $\theta$)
  else if COMPOUND?($x$) and COMPOUND?($y$) then
    return UNIFY(ARGS($x$), ARG($y$), UNIFY(OP($x$), OP($y$, $\theta$))
  else if LIST?($x$) and LIST?($y$) then
    return UNIFY(REST($x$), REST($y$), UNIFY(FIRST($x$), FIRST($y$, $\theta$))
  else return failure

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

Recursively explore two expressions $x$ and $y$ “side by side” to build up a unifier.
Unification Algorithm

function UNIFY(x, y, θ=empty) returns a substitution to make x and y identical, or failure
if θ = failure then return failure
else if x = y then return θ
else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
else if COMPOUND?(x) and COMPOUND?(y) then
    return UNIFY(ARGS(x), ARGs(y), UNIFY(OP(x), OP(y), θ))
else if LIST?(x) and LIST?(y) then
    return UNIFY(REST(x), REST(y), UNIFY(FIRST(x), FIRST(y), θ))
else return failure

function UNIFY-VAR(var, x, θ) returns a substitution
if {var/val} ∈ θ for some val then return UNIFY(val, x, θ)
else if {x/val} ∈ θ for some val then return UNIFY(var, val, θ)
else if OCCUR-CHECK?(var, x) then return failure // check whether the variable var appears
    // inside the complex term x. match fails if so
else return add {var/x} to θ

Recursively explore two expressions x and y “side by side” to build up a unifier.