Inference in First-Order Logic

Outline

I. Knowledge engineering

II. Propositionalization

III. Unification

* Figures are from the textbook site unless a source is specifically cited.

I. Knowledge Engineering Process

- ♣ Identification of questions and facts.

  What will the KB support and what facts are available?

I. 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).

I. 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?

I. 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.

I. 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.

I. 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.

I. 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).

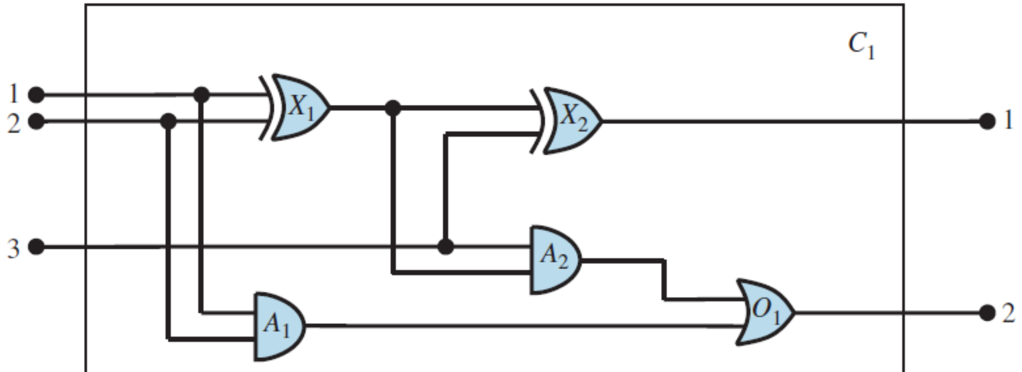
- Formal description of the problem instance.

- 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



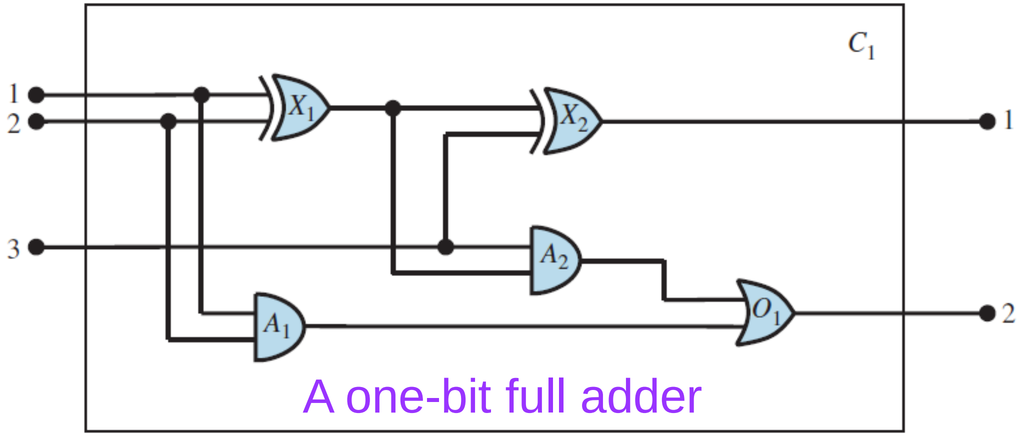
A one-bit full adder

- Four types of gates: *AND*, *OR*, *XOR*, *NOT*.

  1 or 2 inputs, and 1 output

- Represent connections between terminals.

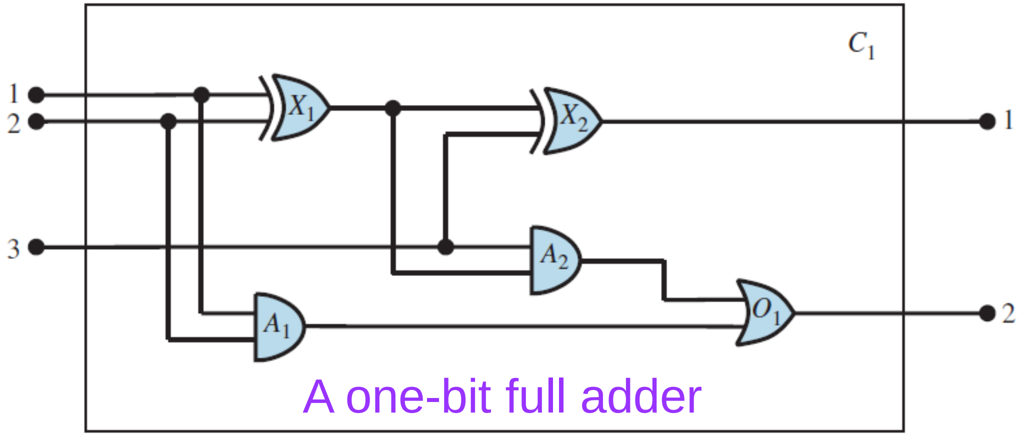
The Electronic Circuits Domain



- 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.

The Electronic Circuits Domain



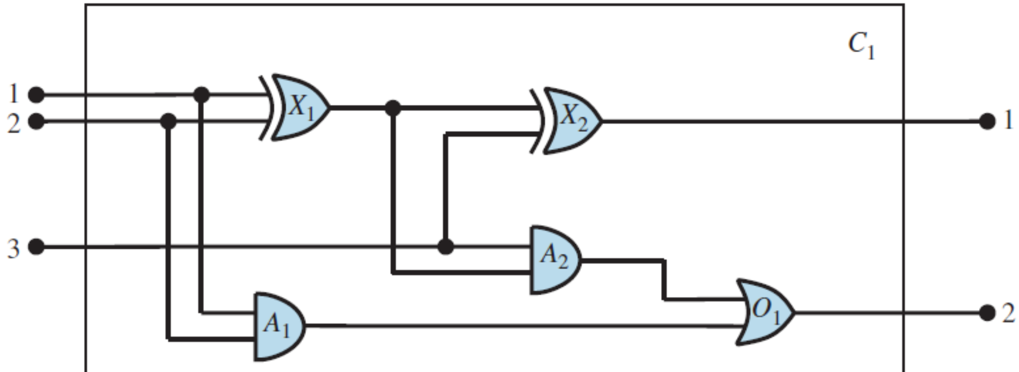
A one-bit full adder

- 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.

The Electronic Circuits Domain



A one-bit full adder

- Four types of gates: *AND*, *OR*, *XOR*, *NOT*.

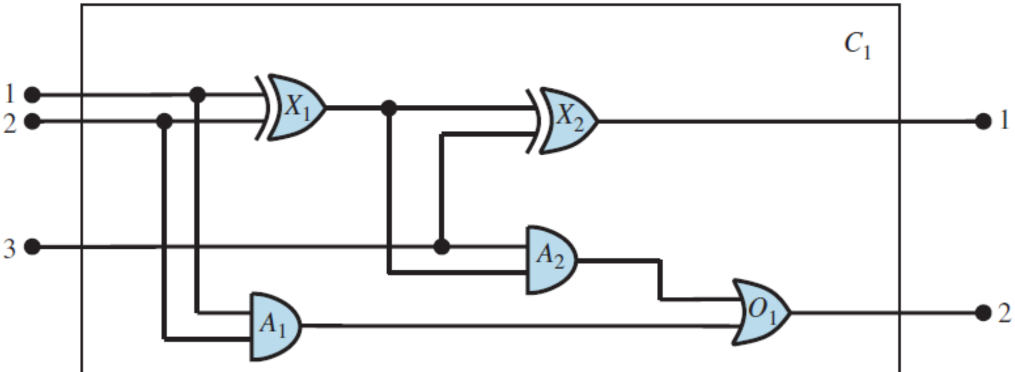
  1 or 2 inputs, and 1 output

- 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*

The Electronic Circuits Domain



A one-bit full adder

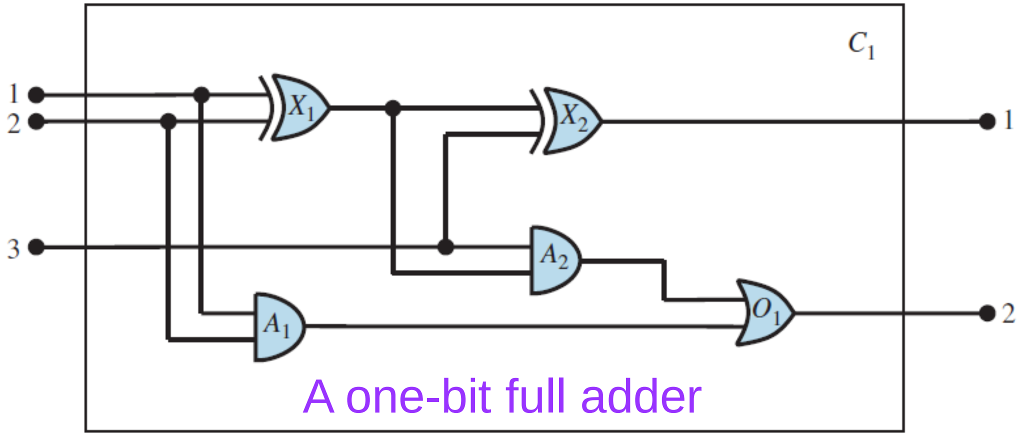
- Four types of gates: *AND*, *OR*, *XOR*, *NOT*.

  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

The Electronic Circuits Domain

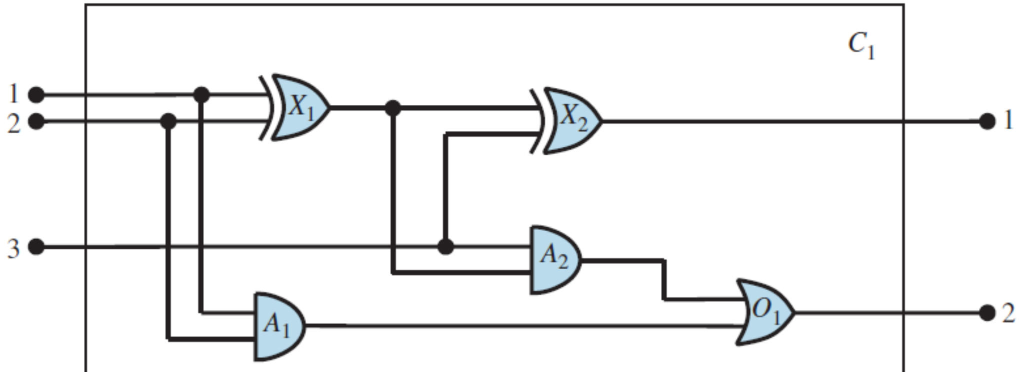


- Four types of gates: *AND*, *OR*, *XOR*, *NOT*.

  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*$\big($*Out*$(1, X_1)$, *In*$(1, X_2)\big)$

The Electronic Circuits Domain



A one-bit full adder

- 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*$\big($*Out*$(1, X_1)$, *In*$(1, X_2)\big)$

♦ Signal function: *Signal*(t) has value 1 or 0 at time t.

Encoding General Domain Knowledge

// Two connected terminals have the same signal.

$\forall t_1, t_2 \quad \mathit{Terminal}(t_1) \wedge \mathit{Terminal}(t_2) \wedge \mathit{Connected}(t_1, t_2) \Rightarrow \mathit{Signal}(t_1) = \mathit{Signal}(t_2)$

Encoding General Domain Knowledge

// Two connected terminals have the same signal.

$\forall t_1, t_2 \;\; \mathit{Terminal}(t_1) \wedge \mathit{Terminal}(t_2) \wedge \mathit{Connected}(t_1, t_2) \Rightarrow \mathit{Signal}(t_1) = \mathit{Signal}(t_2)$

// Every terminal has signal that is either 1 or 0.

$\forall t \;\; \mathit{Terminal}(t) \Rightarrow \mathit{Signal}(\mathrm{t}) = 1 \vee \mathit{Signal}(t) = 0$

Encoding General Domain Knowledge

// Two connected terminals have the same signal.

$\forall t_1, t_2 \quad \mathit{Terminal}(t_1) \wedge \mathit{Terminal}(t_2) \wedge \mathit{Connected}(t_1, t_2) \Rightarrow \mathit{Signal}(t_1) = \mathit{Signal}(t_2)$

// Every terminal has signal that is either 1 or 0.

$\forall t \quad \mathit{Terminal}(t) \Rightarrow \mathit{Signal}(\mathrm{t}) = 1 \vee \mathit{Signal}(t) = 0$

// Connectivity is commutative.

$\forall t_1, t_2 \quad \mathit{Connected}(t_1, t_2) \Leftrightarrow \mathit{Connected}(t_2, t_1)$

Encoding General Domain Knowledge

// Two connected terminals have the same signal.

$\forall t_1, t_2 \quad \mathit{Terminal}(t_1) \land \mathit{Terminal}(t_2) \land \mathit{Connected}(t_1, t_2) \Rightarrow \mathit{Signal}(t_1) = \mathit{Signal}(t_2)$

// Every terminal has signal that is either 1 or 0.

$\forall t \quad \mathit{Terminal}(t) \Rightarrow \mathit{Signal}(\mathrm{t}) = 1 \lor \mathit{Signal}(t) = 0$

// Connectivity is commutative.

$\forall t_1, t_2 \quad \mathit{Connected}(t_1, t_2) \Leftrightarrow \mathit{Connected}(t_2, t_1)$

// Four types of gates.

$\forall g, k \quad \mathit{Gate}(g) \land k = \mathit{Type}(g) \Rightarrow k = \mathit{AND} \lor k = \mathit{OR} \lor k = \mathit{XOR} \lor k = \mathit{NOT}$

Encoding General Domain Knowledge

// Two connected terminals have the same signal.

$\forall t_1, t_2 \;\; \mathit{Terminal}(t_1) \wedge \mathit{Terminal}(t_2) \wedge \mathit{Connected}(t_1, t_2) \Rightarrow \mathit{Signal}(t_1) = \mathit{Signal}(t_2)$

// Every terminal has signal that is either 1 or 0.

$\forall t \;\; \mathit{Terminal}(t) \Rightarrow \mathit{Signal}(\mathrm{t}) = 1 \vee \mathit{Signal}(t) = 0$

// Connectivity is commutative.

$\forall t_1, t_2 \;\; \mathit{Connected}(t_1, t_2) \Leftrightarrow \mathit{Connected}(t_2, t_1)$

// Four types of gates.

$\forall g, k \;\; \mathit{Gate}(g) \wedge k = \mathit{Type}(g) \Rightarrow k = \mathit{AND} \vee k = \mathit{OR} \vee k = \mathit{XOR} \vee k = \mathit{NOT}$

// An AND gate outputs 0 if and only if its input is 0.

$\forall g \;\; \mathit{Gate}(g) \wedge \mathit{Type}(g) = \mathit{AND} \Rightarrow (\mathit{Signal}\big(\mathit{Out}(1, g)\big) = 0 \Leftrightarrow \exists n \;\; \mathit{Signal}\big(\mathit{In}(n, g)\big) = 0)$

Encoding General Domain Knowledge

// Two connected terminals have the same signal.

$\forall t_1, t_2 \;\; \mathit{Terminal}(t_1) \land \mathit{Terminal}(t_2) \land \mathit{Connected}(t_1, t_2) \Rightarrow \mathit{Signal}(t_1) = \mathit{Signal}(t_2)$

// Every terminal has signal that is either 1 or 0.

$\forall t \;\; \mathit{Terminal}(t) \Rightarrow \mathit{Signal}(\mathrm{t}) = 1 \lor \mathit{Signal}(t) = 0$

// Connectivity is commutative.

$\forall t_1, t_2 \;\; \mathit{Connected}(t_1, t_2) \Leftrightarrow \mathit{Connected}(t_2, t_1)$

// Four types of gates.

$\forall g, k \;\; \mathit{Gate}(g) \land k = \mathit{Type}(g) \Rightarrow k = \mathit{AND} \lor k = \mathit{OR} \lor k = \mathit{XOR} \lor k = \mathit{NOT}$

// An AND gate outputs 0 if and only if its input is 0.

$\forall g \;\; \mathit{Gate}(g) \land \mathit{Type}(g) = \mathit{AND} \Rightarrow (\mathit{Signal}(\mathit{Out}(1, g)) = 0 \Leftrightarrow \exists n \;\; \mathit{Signal}(\mathit{In}(n, g)) = 0)$

// An OR gate outputs 1 if and only if any of its input is 1.

$\forall g \;\; \mathit{Gate}(g) \land \mathit{Type}(g) = \mathit{OR} \Rightarrow (\mathit{Signal}(\mathit{Out}(1, g)) = 1 \Leftrightarrow \exists n \;\; \mathit{Signal}(\mathit{In}(n, g)) = 1)$

cont'd

// An XOR gate outputs 1 if and only if its inputs are different.

$$\forall g \quad Gate(g) \wedge Type(g) = XOR \Rightarrow (Signal(Out(1,g)) = 1 \Leftrightarrow Signal(In(1,g)) \neq Signal(In(2,g)))$$

// An NOT gate's output is different from its input.

$$\forall g \quad Gate(g) \wedge Type(g) = NOT \Rightarrow (Signal(Out(1,g)) = 1 \Leftrightarrow Signal(Out(1,g)) \neq Signal(In(1,g)))$$

// All the gates (except for NOT) have two inputs and one output.

$$\forall g \quad Gate(g) \wedge Type(g) = NOT \Rightarrow Arity(g,1,1)$$

$$\forall g,k \quad Gate(g) \wedge k{=}Type(g) \wedge (k = AND \vee k = OR \vee k = XOR) \Rightarrow Arity(g,2,1)$$

// A circuit has terminals exactly up to its input and output arity.

$$\begin{gathered}\forall c,i,j \quad Circuit(c) \wedge Arity(c,i,j) \Rightarrow \\ \forall n \ (n \leq i \Rightarrow Terminal(In(n,c)) \wedge (n > i \Rightarrow In(n,c) = Nothing)) \wedge \\ \forall n \ (n \leq j \Rightarrow Terminal(Out(n,c)) \wedge (n > j \Rightarrow Out(n,c) = Nothing))\end{gathered}$$

// Gates and terminals are all distinct.

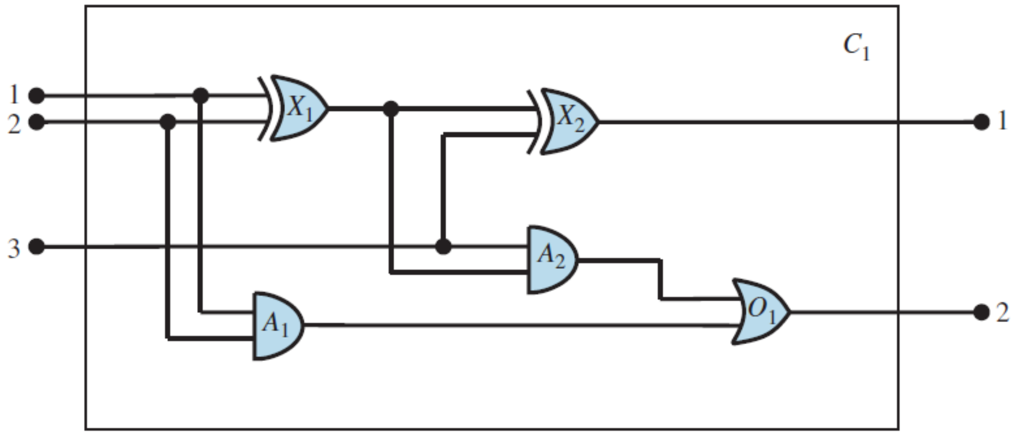
$$\forall g,t,\cancel{s} \quad Gate(g) \wedge Terminal(t) \cancel{\wedge Signal(s)} \Rightarrow g \neq t \cancel{\wedge g \neq s \wedge t \neq s}$$

function not predicate

// Gates are circuits.

$$\forall g \quad Gate(g) \Rightarrow Circuit(g)$$

Encoding a Problem Instance



- Circuit and component gates:

  $Circuit(C_1) \wedge Arity(C_1, 3, 2)$

  $Gate(X_1) \wedge Type(X_1) = XOR$

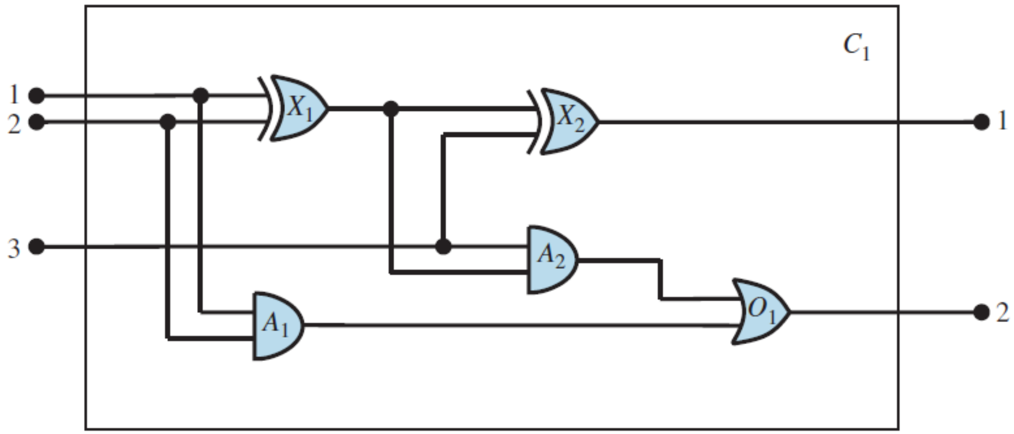
  $Gate(X_2) \wedge Type(X_2) = XOR$

  $Gate(A_1) \wedge Type(A_1) = AND$

  $Gate(A_2) \wedge Type(A_2) = AND$

  $Gate(O_1) \wedge Type(O_1) = OR$

Encoding a Problem Instance



- Circuit and component gates:

  $Circuit(C_1) \land Arity(C_1, 3, 2)$

  $Gate(X_1) \land Type(X_1) = XOR$

  $Gate(X_2) \land Type(X_2) = XOR$

  $Gate(A_1) \land Type(A_1) = AND$

  $Gate(A_2) \land Type(A_2) = AND$

  $Gate(O_1) \land Type(O_1) = OR$

- Connections between the circuit and component gates:

  $Connected(Out(1, X_1), In(1, X_2))$

  $Connected(Out(1, X_1), In(2, A_2))$

  $Connected(Out(1, A_2), In(1, O_1))$

  $Connected(Out(1, A_1), In(2, O_1))$

  $Connected(Out(1, X_2), Out(1, C_1))$

  $Connected(Out(1, O_1), Out(2, C_1))$

  $Connected(In(1, C_1), In(1, X_1))$

  $Connected(In(1, C_1), In(1, A_1))$

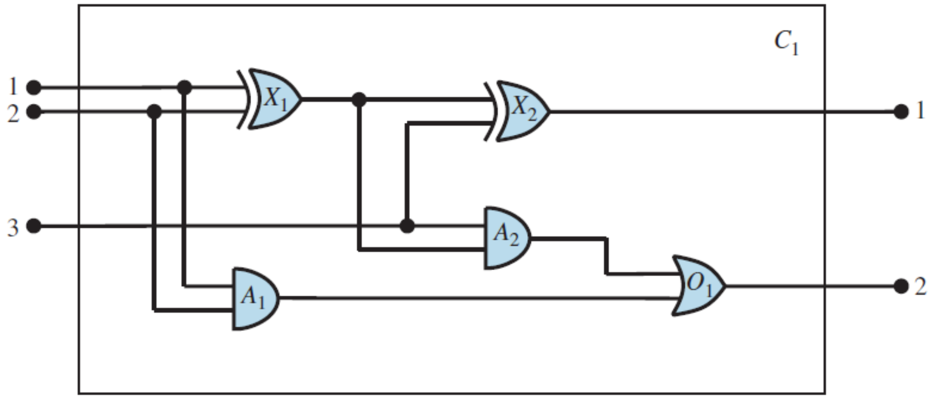
  $Connected(In(2, C_1), In(2, X_1))$

  $Connected(In(2, C_1), In(2, A_1))$

  $Connected(In(3, C_1), In(2, X_2))$

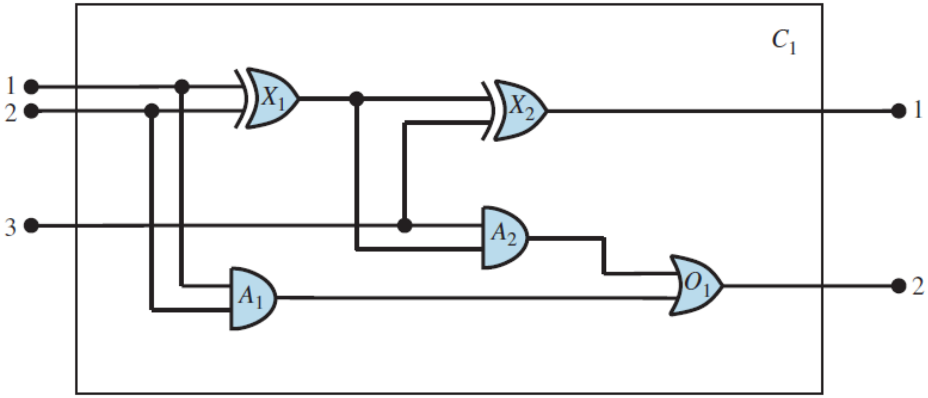
  $Connected(In(3, C_1), In(1, A_2))$

Queries



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

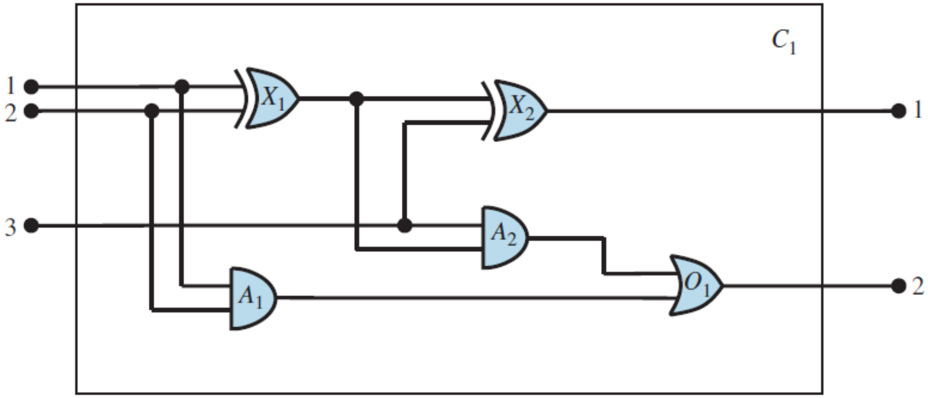
Queries



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

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

Queries



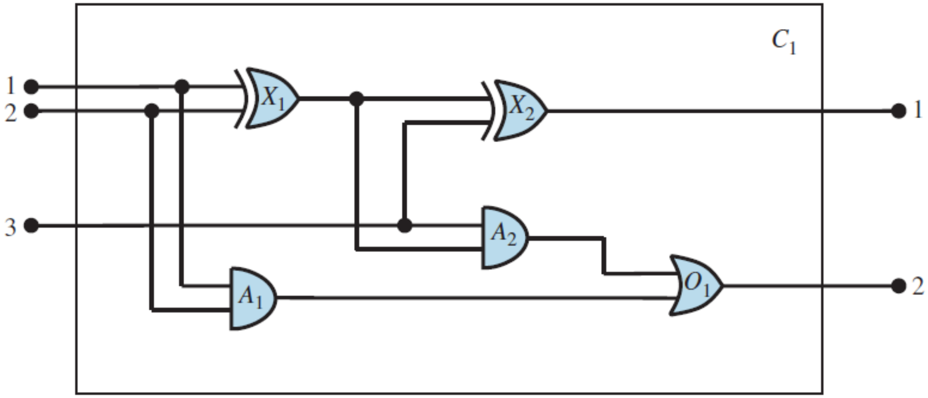
Q. What combinations of inputs would cause the first output of C_1 to be 0 and its second output to be 1?

$$\exists i_1, i_2, i_3 \ \ \mathit{Signal}\big(\mathit{In}(1, C_1)\big) = i_1 \wedge \mathit{Signal}\big(\mathit{In}(2, C_1)\big) = i_2 \wedge \mathit{Signal}\big(\mathit{In}(3, C_1)\big) = i_3 \\ \wedge \mathit{Signal}\big(\mathit{Out}(1, C_1)\big) = 0 \wedge \mathit{Signal}\big(\mathit{Out}(2, C_1)\big) = 1$$

ASKVARS will give three substitutions as answers.

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

Queries



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

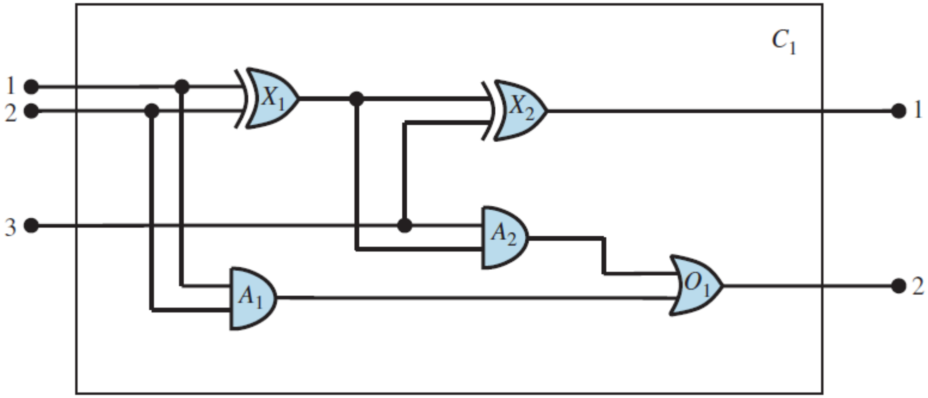
$$\exists i_1, i_2, i_3 \ \mathit{Signal}(\mathit{In}(1, C_1)) = i_1 \wedge \mathit{Signal}(\mathit{In}(2, C_1)) = i_2 \wedge \mathit{Signal}(\mathit{In}(3, C_1)) = i_3 \wedge \mathit{Signal}(\mathit{Out}(1, C_1)) = 0 \wedge \mathit{Signal}(\mathit{Out}(2, C_1)) = 1$$

AskVars will give three substitutions as answers.

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

Debugging: We can also perturb the KB to see what erroneous behaviors would emerge, and then identify missing rules for instance.

Queries



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

$$\exists i_1, i_2, i_3 \ \mathit{Signal}(\mathit{In}(1, C_1)) = i_1 \wedge \mathit{Signal}(\mathit{In}(2, C_1)) = i_2 \wedge \mathit{Signal}(\mathit{In}(3, C_1)) = i_3$$
$$\wedge\, \mathit{Signal}(\mathit{Out}(1, C_1)) = 0 \wedge \mathit{Signal}(\mathit{Out}(2, C_1)) = 1$$

AskVars will give three substitutions as answers.

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

Debugging: We can also perturb the KB to see what erroneous behaviors would emerge, and then identify missing rules for instance.

II. Propositional vs. First-Order Inference

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

II. Propositional vs. First-Order Inference

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

- Eliminate universal quantifiers.

$\forall x\;\; \textit{Human}(x) \Rightarrow \textit{Fallible}(x)$ // All humans are fallible.

II. Propositional vs. First-Order Inference

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

- Eliminate universal quantifiers.

$$\forall x \;\; \mathit{Human}(x) \Rightarrow \mathit{Fallible}(x)$$ // All humans are fallible.

We can infer sentences like

$$\mathit{Human}(\mathit{Socrates}) \Rightarrow \mathit{Fallible}(\mathit{Socrates})$$
$$\mathit{Human}(\mathit{Einstein}) \Rightarrow \mathit{Fallible}(\mathit{Einstein})$$
$$\mathit{Human}(\mathit{Messi}) \Rightarrow \mathit{Fallible}(\mathit{Messi})$$
$$\vdots$$

II. 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

$$Human(Socrates) \Rightarrow Fallible(Socrates)$$
$$Human(Einstein) \Rightarrow Fallible(Einstein)$$
$$Human(Messi) \Rightarrow Fallible(Messi)$$
$$\vdots$$

From

// All birds are warm-blooded and have wings.

$$\forall x \ \ Bird(x) \Rightarrow WarmBlooded(x) \wedge HaveWings(x)$$

II. 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

$Human(Socrates) \Rightarrow Fallible(Socrates)$

$Human(Einstein) \Rightarrow Fallible(Einstein)$

$Human(Messi) \Rightarrow Fallible(Messi)$

$\vdots$

From

// All birds are warm-blooded and have wings.

$\forall x \; Bird(x) \Rightarrow WarmBlooded(x) \wedge HaveWings(x)$

We can infer (if *Bird*(*Ostrich*) and *Bird*(*Peacock*) are in the KB):

$WarmBlooded(Ostrich) \wedge HaveWings(Ostrich)$

$WarmBlooded(Peacock) \wedge HaveWings(Peacock)$

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.

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

Universal and Existential Instantiations

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

Substitute a ground term for a universally quantified variable.

sentence

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

Universal and Existential Instantiations

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

Substitute a ground term for a universally quantified variable.

$$\frac{\overset{\text{sentence}}{\forall v\ \alpha}}{\textsc{Subst}(\underbrace{\{v/g\}}_{\text{substitution } \theta,\text{ e.g., } \theta = \{x/\mathit{Socrates}\}}, \alpha)}$$

Universal and Existential Instantiations

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

Substitute a ground term for a universally quantified variable.

sentence

$$\frac{\forall v\ \alpha}{\textsc{Subst}(\underbrace{\{v/g\}}_{\text{substitution } \theta}, \alpha)}$$

substitution θ, e.g., $\theta = \{x/\mathit{Socrates}\}$

E.g., $\theta = \{x/\mathit{Ostrich}\}$

$\forall x\ \ \mathit{Bird}(x) \Rightarrow \mathit{WarmBlooded}(x) \wedge \mathit{HaveWings}(x)$

Universal and Existential Instantiations

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

Substitute a ground term for a universally quantified variable.

$$\frac{\overset{\text{sentence}}{\forall v\ \alpha}}{\textsc{Subst}(\underbrace{\{v/g\}}_{\text{substitution } \theta\text{, e.g., } \theta = \{x/Socrates\}}, \alpha)}$$

E.g., $\theta = \{x/\mathit{Ostrich}\}$

$$\forall x\ \overbrace{\mathit{Bird}(x) \Rightarrow \mathit{WarmBlooded}(x) \wedge \mathit{HaveWings}(x)}^{\alpha}$$

Universal and Existential Instantiations

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

Substitute a ground term for a universally quantified variable.

$$\frac{\overset{\text{sentence}}{\forall v\ \alpha}}{\textsc{Subst}(\underbrace{\{v/g\}}_{\text{substitution } \theta\text{, e.g., } \theta = \{x/\mathit{Socrates}\}}, \alpha)}$$

E.g., $\theta = \{x/\mathit{Ostrich}\}$

$$\frac{\forall x\ \overbrace{\mathit{Bird}(x) \Rightarrow \mathit{WarmBlooded}(x) \wedge \mathit{HaveWings}(x)}^{\alpha}}{}$$

$\textsc{Subst}(\theta, \alpha) \quad \equiv \quad \mathit{Bird}(\mathit{Ostrich}) \Rightarrow \mathit{WarmBlooded}(\mathit{Ostrich}) \wedge \mathit{HaveWings}(\mathit{Ostrich})$

Existential Instantiation

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

$$\frac{\exists v\ \alpha}{\textsc{Subst}(\{v/g\}, \alpha)}$$

Existential Instantiation

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

$$\frac{\exists v \ \alpha}{\textsc{Subst}(\{v/g\}, \alpha)}$$

From

$$\exists y \ \ \mathit{Mother}(y, \mathit{Liam})$$

Existential Instantiation

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

$$\frac{\exists v\ \alpha}{\textsc{Subst}(\{v/g\}, \alpha)}$$

From

$$\exists y\ \ \mathit{Mother}(y, \mathit{Liam})$$

we can infer

$$\mathit{Mother}(\mathit{LiamsMom}, \mathit{Liam})$$

as long as *LiamsMom* does **not** appear elsewhere in the KB.

Existential Instantiation

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

$$\frac{\exists v\ \alpha}{\textsc{Subst}(\{v/g\}, \alpha)}$$

From

$$\exists y\ \ \mathit{Mother}(y, \mathit{Liam})$$

we can infer

$$\mathit{Mother}(\mathit{LiamsMom}, \mathit{Liam})$$

as long as *LiamsMom* does **not** appear elsewhere in the KB.

Skolem constant

Propositionalization

$$\forall x \forall y \quad Ancestor(x, y) \rightarrow Parent(x, y) \vee \exists z (Ancestor(x, z) \wedge Ancestor(z, y))$$

KB:

$Ancestor(John, David)$

$Parent(John, David)$

$Parent(David, Lisa)$

Propositionalization

$$\forall x \forall y \quad Ancestor(x, y) \rightarrow Parent(x, y) \vee \exists z (Ancestor(x, z) \wedge Ancestor(z, y))$$

KB:

$Ancestor(John, David)$

$Parent(John, David)$

$Parent(David, Lisa)$

$$Ancestor(John, David)$$
$$\rightarrow Parent(John, David) \vee (Ancestor(John, JohnDavidAnc) \wedge Ancestor(JohnDavidAnc, David))$$

Propositionalization

$$\forall x \forall y \quad \mathit{Ancestor}(x, y) \rightarrow \mathit{Parent}(x, y) \vee \exists z (\mathit{Ancestor}(x, z) \wedge \mathit{Ancestor}(z, y))$$

KB:

$\mathit{Ancestor}(\mathit{John}, \mathit{David})$

$\mathit{Parent}(\mathit{John}, \mathit{David})$

$\mathit{Parent}(\mathit{David}, \mathit{Lisa})$

$$\mathit{Ancestor}(\mathit{John}, \mathit{David}) \rightarrow \mathit{Parent}(\mathit{John}, \mathit{David}) \vee (\mathit{Ancestor}(\mathit{John}, \mathit{JohnDavidAnc}) \wedge \mathit{Ancestor}(\mathit{JohnDavidAnc}, \mathit{David}))$$

$$\mathit{Ancestor}(\mathit{David}, \mathit{John}) \rightarrow \mathit{Parent}(\mathit{John}, \mathit{David}) \vee (\mathit{Ancestor}(\mathit{David}, \mathit{DavidJohnAnc}) \wedge \mathit{Ancestor}(\mathit{DavidJohnAnc}, \mathit{David}))$$

$$\mathit{Ancestor}(\mathit{John}, \mathit{Lisa}) \rightarrow \mathit{Parent}(\mathit{John}, \mathit{Lisa}) \vee (\mathit{Ancestor}(\mathit{John}, \mathit{JohnLisaAnc}) \wedge \mathit{Ancestor}(\mathit{JohnLisaAnc}, \mathit{Lisa}))$$

$$\vdots$$

Skolemization

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

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.

$$\exists y \;\; P(y, x_1, \ldots\ldots, x_n)$$

// y depends on $x_1, \ldots\ldots, x_n$

Skolemization

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.

$$\exists y \; P(y, x_1, \ldots., x_n)$$ // y depends on $x_1, \ldots., x_n$

$\Downarrow$ eliminate y by introducing function f

$$P(f(x_1, \ldots., x_n), x_1, \ldots., x_n)$$

Skolemization

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.

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

$$\Downarrow \text{ eliminate } y \text{ by introducing function } f$$

$$P(f(x_1, \ldots, x_n), x_1, \ldots, x_n)$$

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

Skolemization

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.

$\exists y \;\; P(y, x_1, \ldots., x_n)$ // y depends on $x_1, \ldots., x_n$

$\Downarrow$ eliminate y by introducing function f

$$P(f(x_1, \ldots., x_n), x_1, \ldots., x_n)$$

- That $P(y, x_1, \ldots., x_n) = \mathit{true}$ 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

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.

$$\exists y \ P(y, x_1, \ldots, x_n)$$ // y depends on $x_1, \ldots, x_n$

$\Downarrow$ eliminate y by introducing function f

$$P(f(x_1, \ldots, x_n), x_1, \ldots, x_n)$$

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

$\exists y \ \ Mother(y, Liam)$

$\exists y \ \ Mother(y, Sophia)$

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

$Mother(mom(Liam), Liam)$

$Mother(mom(Sophia), Sophia)$

Skolemization

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.

$$\exists y \;\; P(y, x_1, \ldots, x_n)$$ // y depends on $x_1, \ldots, x_n$

$\Downarrow$ eliminate y by introducing function f

$$P(f(x_1, \ldots, x_n), x_1, \ldots, x_n)$$

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

$\exists y \;\; \textit{Mother}(y, \textit{Liam})$

$\exists y \;\; \textit{Mother}(y, \textit{Sophia})$

$\xrightarrow{\text{new unary function } \textit{mom}()}$

$\textit{Mother}(\textit{mom}(\textit{Liam}), \textit{Liam})$

$\textit{Mother}(\textit{mom}(\textit{Sophia}), \textit{Sophia})$

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

Generalized Modus Ponens

$$(p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q), \quad p_1', p_2', \ldots, p_n'$$

Suppose there exists a substitution θ such that

$$\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i') \qquad \text{for } 1 \leq i \leq n$$

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 θ such that

$$\textsc{Subst}(\theta, p_i) = \textsc{Subst}(\theta, p_i') \qquad \text{for } 1 \le i \le n$$

Then

$$\frac{p_1', p_2', \ldots, p_n', \quad (p_1 \land p_2 \land \cdots \land p_n \Rightarrow q)}{\textsc{Subst}(\theta, q)}$$

Generalized Modus Ponens

$$(p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q), \quad p_1', p_2', \ldots, p_n'$$

Suppose there exists a substitution θ such that

$$\textsc{Subst}(\theta, p_i) = \textsc{Subst}(\theta, p_i') \qquad \text{for } 1 \le i \le n$$

Then

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

KB:

$\mathit{Gate}(X_1)$, $\mathit{Terminal}(\mathit{In}(1, C_1))$ $\qquad$ $\mathit{Gate}(g) \wedge \mathit{Terminal}(t) \Rightarrow g \neq t$

Generalized Modus Ponens

$$(p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q), \quad p_1', p_2', \ldots, p_n'$$

Suppose there exists a substitution θ such that

$$\textsc{Subst}(\theta, p_i) = \textsc{Subst}(\theta, p_i') \qquad \text{for } 1 \leq i \leq n$$

Then

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

KB:

$Gate(X_1)$, $Terminal(In(1, C_1))$ $\qquad Gate(g) \wedge Terminal(t) \Rightarrow g \neq t$

$\textsc{Subst}(\theta, q)$ $\qquad \theta = \{g/X_1, t/(In(1, C_1))\}$

q is $g \neq t$

Generalized Modus Ponens

$$(p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q), \quad p_1', p_2', \ldots, p_n'$$

Suppose there exists a substitution θ such that

$$\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i') \qquad \text{for } 1 \leq i \leq n$$

Then

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

KB:

$\mathit{Gate}(X_1)$, $\mathit{Terminal}(\mathit{In}(1, C_1))$ $\qquad$ $\mathit{Gate}(g) \wedge \mathit{Terminal}(t) \Rightarrow g \neq t$

$\text{SUBST}(\theta, q)$ $\Downarrow$ $\theta = \{g/X_1, t/(\mathit{In}(1, C_1))\}$

q is $g \neq t$

$$X_1 \neq \mathit{In}(1, C_1)$$

III. Unification

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

- Carried out by the algorithm UNIFY.

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

III. Unification

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

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

sentences

III. Unification

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

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

sentences

Query: $AskVars(Knows(John, x))$ // what does John know?

III. Unification

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

- Carried out by the algorithm UNIFY.

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

sentences

Query: $AskVars(Knows(John, x))$ // what does John know?

Answers: all the sentences in the KB found to unify with $Knows(John, x)$.

III. Unification

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

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

sentences

Query: *AskVars*(*Knows*(*John*, x)) // what does John know?

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

UNIFY(*Knows*(*John*, x), *Knows*(*John*, *Jane*)) = {x/*Jane*}

UNIFY(*Knows*(*John*, x), *Knows*(y, *Bill*)) = {x/*Bill*, y/*John*}

UNIFY(*Knows*(*John*, x), *Knows*(y, *Mother*(y))) = {y/*John*, x/*Mother*(*John*)}

Unification (cont'd)

- Conflicting substitutions

$\text{U}\text{NIFY}(\mathit{Knows}(\mathit{John}, x), \mathit{Knows}(x, \mathit{Elizabeth})) = \mathit{failure}$

Unification (cont'd)

- Conflicting substitutions

$\text{U}\small{\text{NIFY}}(\textit{Knows}(\textit{John}, x), \textit{Knows}(x, \textit{Elizabeth})) = \textit{failure}$

$\{x/\textit{John}\}$

Unification (cont'd)

♠ Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*

$\{x/\mathit{John}\}$ $\{x/\mathit{Elizabeth}\}$

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*

$\{x/John\}$ $\{x/Elizabeth\}$

x cannot take on the values *John* and *Elizabeth* at the same time!

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*

{x/*John*}   {x/*Elizabeth*}

x cannot take on the values *John* and *Elizabeth* at the same time!

- Multiple unifiers

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

could return {y/*John*, x/z}

or {y/*John*, x/*John*, z/*John*}

Unification (cont'd)

♠ Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*

$\{x/John\}$ $\{x/Elizabeth\}$

x cannot take on the values *John* and *Elizabeth* at the same time!

♠ Multiple unifiers

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

could return $\{y/John,\ x/z\}$ $\Longrightarrow$ *Knows*(*John*, z)

or $\{y/John,\ x/John,\ z/John\}$

Unification (cont'd)

- Conflicting substitutions

$\textsc{Unify}(\mathit{Knows}(\mathit{John}, x), \mathit{Knows}(x, \mathit{Elizabeth})) = \mathit{failure}$

$\{x/\mathit{John}\}$ $\{x/\mathit{Elizabeth}\}$

x cannot take on the values *John* and *Elizabeth* at the same time!

- Multiple unifiers

$\textsc{Unify}(\mathit{Knows}(\mathit{John}, x), \mathit{Knows}(y, z))$

could return $\{y/\mathit{John}, x/z\}$ $\Longrightarrow$ $\mathit{Knows}(\mathit{John}, z)$

or $\{y/\mathit{John}, x/\mathit{John}, z/\mathit{John}\}$ $\Longrightarrow$ $\mathit{Knows}(\mathit{John}, \mathit{John})$

Unification (cont'd)

- Conflicting substitutions

$\text{UNIFY}(Knows(John, x), Knows(x, Elizabeth))$ = *failure*

$\{x/John\}$ $\{x/Elizabeth\}$

x cannot take on the values *John* and *Elizabeth* at the same time!

- Multiple unifiers

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

could return $\{y/John, x/z\}$ $\Longrightarrow$ $Knows(John, z)$

more general unifier for fewer restriction on variable values

or $\{y/John, x/John, z/John\}$ $\Longrightarrow$ $Knows(John, John)$

Unification (cont'd)

♠ Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*

{x/*John*}   {x/*Elizabeth*}

x cannot take on the values *John* and *Elizabeth* at the same time!

♠ Multiple unifiers

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

could return {y/*John*, x/z} ⟹ *Knows*(*John*, z)

more general unifier for fewer restriction on variable values

or {y/*John*, x/*John*, z/*John*} ⟹ *Knows*(*John*, *John*)

less general unifier

Unification (cont'd)

- Conflicting substitutions

UNIFY(*Knows*(*John*, x), *Knows*(x, *Elizabeth*)) = *failure*

{x/*John*}　　　　{x/*Elizabeth*}

x cannot take on the values *John* and *Elizabeth* at the same time!

- Multiple unifiers

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

could return {y/*John*, x/z} $\Longrightarrow$ *Knows*(*John*, z)

more general unifier for fewer restriction on variable values

or {y/*John*, x/*John*, z/*John*} $\Longrightarrow$ *Knows*(*John*, *John*)

less general 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), $\boxed{\text{ARGS}(y)}$, UNIFY($\boxed{\text{OP}(x)}$, 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 symbol of x

argument list of y

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

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), $\boxed{\text{ARGS}(y)}$, UNIFY($\boxed{\text{OP}(x)}$, 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 symbol of x

argument list of y

function UNIFY-VAR(var, x, θ) **returns** a substitution
  if $\{var/val\} \in \theta$ for some val **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ for some val **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** *failure* // check whether the variable *var* appears
  else return add $\{var/x\}$ to θ // inside the complex term x. match fails if so
// because no unifier can be constructed.

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), $\boxed{\text{ARGS}(y)}$, UNIFY($\boxed{\text{OP}(x)}$, 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 symbol of x

argument list of y

function UNIFY-VAR(var, x, θ) **returns** a substitution
  if $\{var/val\} \in \theta$ for some val **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ for some val **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** *failure* // check whether the variable *var* appears
  else return add $\{var/x\}$ to θ // inside the complex term x. match fails if so
// because no unifier can be constructed.

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