Syllabus:

First Order Logic: Representation Revisited, Syntax and Semantics of First Order logic, Using First Order logic, Knowledge Engineering In First-Order Logic

Inference in First Order Logic: Propositional Versus First Order Inference, Unification ,Forward Chaining

Chapter 8- 8.1, 8.2, 8.3, 8.4

Chapter 9- 9.1, 9.2, 9.3

Text book: Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson, 2015

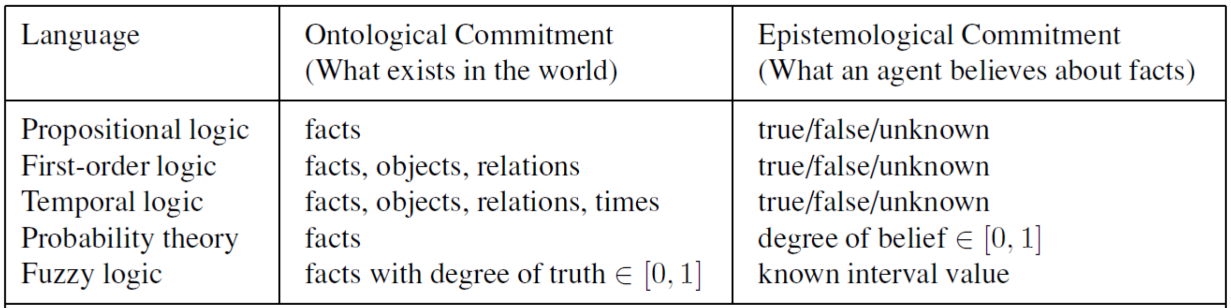
- The language of thought
- Combining the best of natural and formal languages.

Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries . . .

Relations: these can be unary relations or properties such as red, round, bogus, prime,multistoried . . ., or more general n-ary relations such as brother of, bigger than, inside,part of, has color, occurred after, owns, comes between, . . .

Functions: father of, best friend, third inning of, one more than, beginning of .

1. "One plus two equals three."
   - Objects: one, two, three, one plus two; Relation: equals; Function: plus. ("One plus two" is a name for the object that is obtained by applying the function "plus" to the objects "one" and "two." "Three" is another name for this object.)
2. "Squares neighboring the wumpus are smelly."
   - Objects: wumpus, squares; Property: smelly; Relation: neighboring.
3. "Evil King John ruled England in 1200."
   - Objects: John, England, 1200; Relation: ruled; Properties: evil, king



Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value

Figure 8.1 Formal languages and their ontological and epistemological commitments.

Syntax and Semantics of First Order Logic

1. Models for First Order Logic

Models for propositional logic link proposition symbols to predefined truth values.

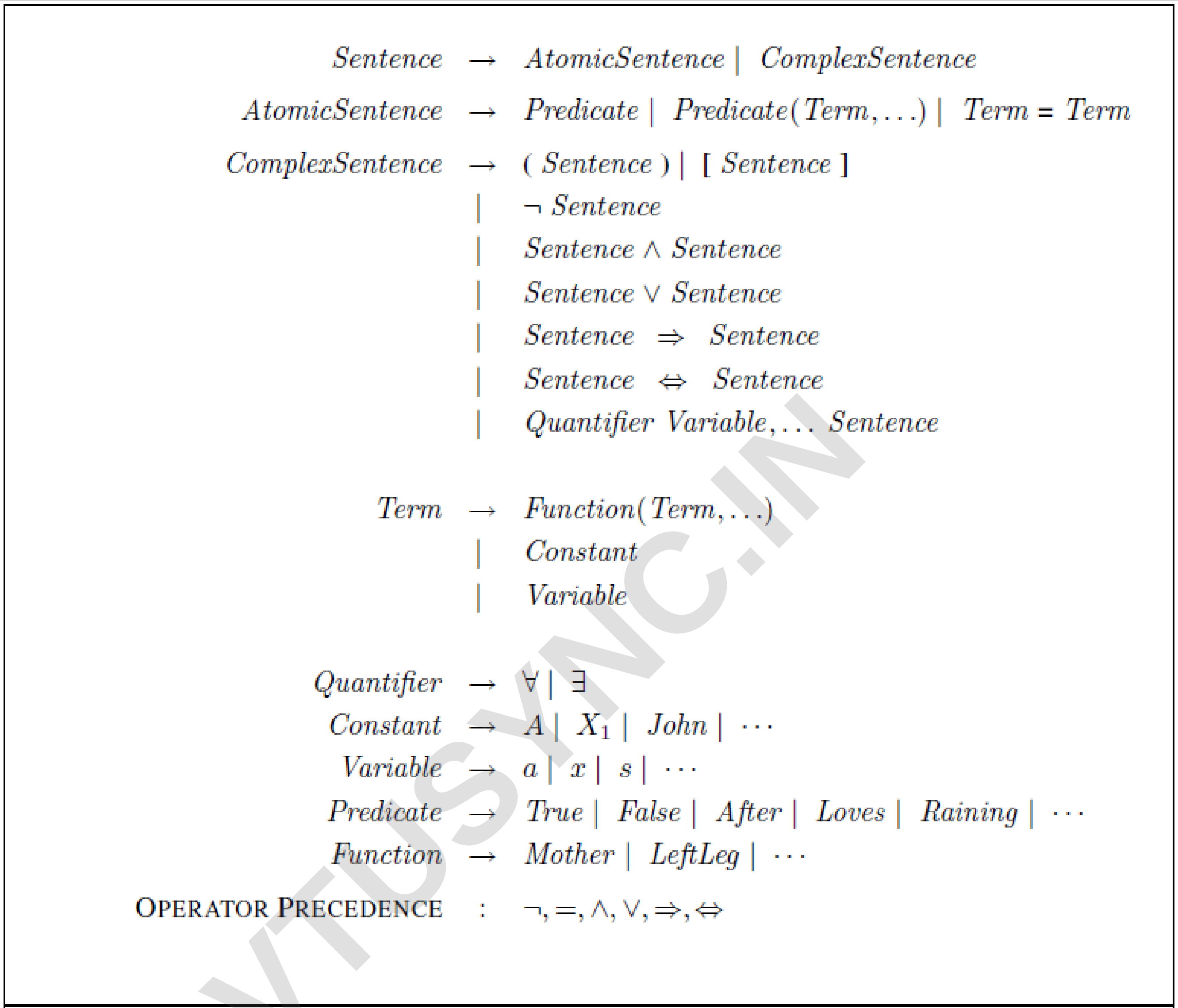
Representation of tuple

2. Symbols and Interpretations

Constant symbols→Objects

Predicate symbols→relations

Function symbols→functions



$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{Predicate} \mid \mathit{Predicate}(\mathit{Term},\ldots) \mid \mathit{Term} = \mathit{Term} \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
 & \mid & \neg\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \Rightarrow \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \Leftrightarrow \mathit{Sentence} \\
 & \mid & \mathit{Quantifier}\ \mathit{Variable},\ldots\ \mathit{Sentence} \\
 & & \\
\mathit{Term} & \rightarrow & \mathit{Function}(\mathit{Term},\ldots) \\
 & \mid & \mathit{Constant} \\
 & \mid & \mathit{Variable} \\
 & & \\
\mathit{Quantifier} & \rightarrow & \forall \mid \exists \\
\mathit{Constant} & \rightarrow & A \mid X_1 \mid \mathit{John} \mid \cdots \\
\mathit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\mathit{Predicate} & \rightarrow & \mathit{True} \mid \mathit{False} \mid \mathit{After} \mid \mathit{Loves} \mid \mathit{Raining} \mid \cdots \\
\mathit{Function} & \rightarrow & \mathit{Mother} \mid \mathit{LeftLeg} \mid \cdots \\
\text{OPERATOR PRECEDENCE} & : & \neg, =, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1060 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

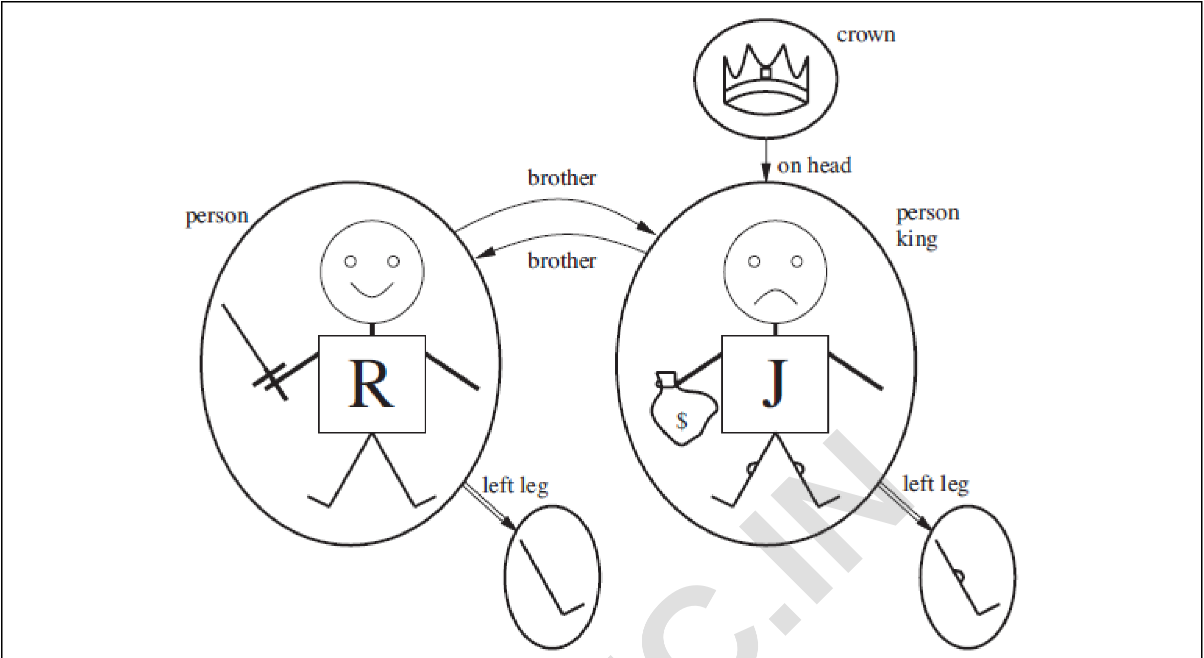


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

Atomic sentence

Brother(Richard,John)

Complex sentence

¬Brother (LeftLeg(Richard), John)

Brother (Richard , John) ∧ Brother (John,Richard) King(Richard) ∨ King(John)

¬King(Richard) ⇒ King(John)

1. First-order logic contains two standard quantifiers, called ***universal*** **and** ***existential***

"All kings are persons"

$\forall$ x King(x) ⇒ Person(x) .

Universal quantification makes statements about every object.

A statement about *some* object in the universe without naming it, by using an existential quantifier.

King John has a crown on his head ∃ x Crown(x) ∧ OnHead(x, John) .

"For all x, if x is a king, then x is a person." The symbol x is called a **variable**.

By convention, variables are lowercase letters.

A variable is a term all by itself, and as such can also serve as the argument of a function—for example, LeftLeg(x).

A term with no variables is called a **ground term.**

- Often want to express more complex sentences using multiple quantifiers.
- The simplest case is where the quantifiers are of the same type. For example, “Brothers are siblings” can be written as

  ∀ x ∀ y Brother (x, y) ⇒ Sibling(x, y) .
- Consecutive quantifiers of the same type can be written as one quantifier with several variables.
- For example, to say that siblinghood is a symmetric

relationship, we can write

∀ x, y Sibling(x, y) ⇔ Sibling(y, x) .

- In other cases we will have mixtures.

“Everybody likes somebody” means

that for every person, there is someone that person likes:

∀ x ∃ y Likes(x, y)

- On the other hand, to say “There is someone who is liked by everyone,” we write

  ∃ y ∀ x Likes(x, y)

- $$\begin{array}{llllll} \forall x\ \neg P & \equiv & \neg\exists x\ P & \neg(P \lor Q) & \equiv & \neg P \land \neg Q \\ \neg\forall x\ P & \equiv & \exists x\ \neg P & \neg(P \land Q) & \equiv & \neg P \lor \neg Q \\ \forall x\ P & \equiv & \neg\exists x\ \neg P & P \land Q & \equiv & \neg(\neg P \lor \neg Q) \\ \exists x\ P & \equiv & \neg\forall x\ \neg P & P \lor Q & \equiv & \neg(\neg P \land \neg Q) \end{array}$$

  First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier.
- We can use the **equality symbol** to signify that two terms refer to the same object. For example, Father (John)=Henry
- says that the object referred to by Father (John) and the object referred to by Henry are the same.
- Because an interpretation fixes the referent of any term, determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object.

1. Assertions and queries in first-order logic

- Sentences are added to a knowledge base using TELL, exactly as in propositional logic.
- Such sentences are called **assertions**. For example, we can assert that John is a king, Richard is a person, and all kings are persons:
- TELL(KB, King(John)) .

- TELL(KB, Person(Richard)) .
- TELL(KB, ∀ x King(x) ⇒ Person(x)) .
- We can ask questions of the knowledge base using ASK.
- For example, ASK(KB, King(John)) returns true.
- Questions asked with ASK are called **queries** or **goals**.

2. The kinship domain

- It is the domain of family relationships, or kinship.
- This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William" and rules such as "One's grandmother is the mother of one's parent."
- Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: Parent, Sibling, Brother , Sister , Child , Daughter, Son, Spouse, Wife, Husband, Grandparent , Grandchild , Cousin, Aunt, and Uncle.
- For example, one's mother is one's female parent:

$$\forall\, m, c\ \text{Mother}(c)=m \Leftrightarrow \text{Female}(m) \land \text{Parent}(m, c)$$

Few more examples:

1. Male and female are disjoint categories:

$$\forall\, x\ \text{Male}(x) \Leftrightarrow \neg\text{Female}(x)\ .$$

2.Parent and child are inverse relations:

$$\forall\, p, c\ \text{Parent}(p, c) \Leftrightarrow \text{Child}(c, p)\ .$$

3. A grandparent is a parent of one's parent:

$$\forall\, g, c\ \text{Grandparent}(g, c) \Leftrightarrow \exists p\ \text{Parent}(g, p) \land \text{Parent}(p, c)\ .$$

4. A sibling is another child of one's parents:

$$\forall\, x, y\ \text{Sibling}(x, y) \Leftrightarrow x = y \land \exists p\ \text{Parent}(p, x) \land \text{Parent}(p, y)\ .$$

3.Numbers,Sets and lists

- The theory of **natural numbers** or non-negative integers.
- Here we need a predicate NatNum that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, S (successor).
- The **Peano axioms** define natural numbers and addition.
- Natural numbers are defined recursively:

$$\text{NatNum}(0)\ . \qquad \forall\, n\ \text{NatNum}(n) \Rightarrow \text{NatNum}(S(n))$$

- That is, 0 is a natural number, and for every object n, if n is a natural number, then S(n) is a natural number. So the natural numbers are 0, S(0), S(S(0)), and so on.

- We also need axioms to constrain the successor function:
- $\forall n\ 0 \neq S(n)$. $\forall m, n\ m \neq n \Rightarrow S(m) \neq S(n)$.
- Now we can define addition in terms of the successor function:

1.$\forall m\ NatNum(m) \Rightarrow +(0,m) = m$.
2.$\forall m, n\ NatNum(m) \land NatNum(n) \Rightarrow +(S(m), n) = S(+(m, n))$.

- The first of these axioms says that adding 0 to any natural number m gives m itself.
- The use of the binary function symbol "+" in the term +(m, 0); in ordinary mathematics, the term would be written m + 0 using infix notation.
- S(n) as n+ 1, so the second axiom becomes
- $\forall m, n\ NatNum(m) \land NatNum(n) \Rightarrow (m + 1) + n = (m + n) + 1$.

- **The empty set is a constant written as {}.**
- **There is one unary predicate, Set, which is true of sets.**
- **The binary predicates are x∈ s (x is a member of set s) and s1 ⊆ s2 (set s1 is a subset, not necessarily proper, of set s2).**
- **The binary functions are s1 ∩ s2 (the intersection of two sets), s1 ∪ s2 (the union of two sets), and {x|s} (the set resulting from adjoining element x to set s). One possible set of axioms is as follows:**

1. The only sets are the empty set and those made by adjoining something to a set:

$\forall s\ Set(s) \Leftrightarrow (s=\{\}) \lor (\exists x, s2\ Set(s2) \land s=\{x|s2\})$.

2. The empty set has no elements adjoined into it. In other words, there is no way to decompose {} into a smaller set and an element:

$\neg\exists x, s\ \{x|s\}=\{\}$.

3. Adjoining an element already in the set has no effect: $\forall x, s\ x\in s \Leftrightarrow s=\{x|s\}$.

4. The only members of a set are the elements that were adjoined into it.

We express this recursively, saying that x is a member of s if and only if s is equal to some set s2 adjoined with some element y, where either y is the same as x or x is a member of s2: $\forall x, s\ x\in s \Leftrightarrow \exists y, s2\ (s=\{y|s2\} \land (x=y \lor x\in s2))$

5. A set is a subset of another set if and only if all of the first set's members are members of the second

set:

$$\forall s1, s2\ s1 \subseteq s2 \Leftrightarrow (\forall x\ x\in s1 \Rightarrow x\in s2)$$

6. Two sets are equal if and only if each is a subset of the other:

$$\forall s1, s2\ (s1 = s2) \Leftrightarrow (s1 \subseteq s2 \land s2 \subseteq s1)$$

7. An object is in the intersection of two sets if and only if it is a member of both sets:

$$\forall x, s1, s2\ x\in (s1 \cap s2) \Leftrightarrow (x\in s1 \land x\in s2)\ .$$

8. An object is in the union of two sets if and only if it is a member of either set:

$$\forall x, s1, s2\ x\in (s1 \cup s2) \Leftrightarrow (x\in s1 \lor x\in s2)\ .$$

- Lists are similar to sets.
- The differences are that lists are ordered and the same element can appear more than once in a list.
- We can use the vocabulary of Lisp for lists: Nil is the constant list with no elements; Cons, Append, First, and Rest are functions; and Find is the predicate that does for lists what Member does for sets. List? is a predicate that is true only of lists.
- The empty list is [ ]. The term Cons(x, y), where y is a nonempty list, is written [x|y].
- The term Cons(x, Nil) (i.e., the list containing the element x) is written as [x].
- A list of several elements, such as [A,B,C], corresponds to the nested term Cons(A, Cons(B, Cons(C, Nil))).

3. The WUMPUS WORLD

The wumpus agent receives a percept vector with five elements.

Percept ([Stench, Breeze, Glitter , None, None]

- The actions in the wumpus world can be represented by logical terms:
- Turn(Right), Turn(Left), Forward , Shoot , Grab, Climb .
- If the agent is at a square and perceives a breeze, then that square is breezy:

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

$$\forall s\ Breezy(s) \Leftrightarrow \exists r\ Adjacent(r, s) \land Pit(r)\ .$$

- <u>**STEPS in Knowledge Engineering Process**</u>

1. **Identify the task**. The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance.

2. Assemble the relevant knowledge. The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know—a process called **knowledge acquisition**.

At this stage, the knowledge is not represented formally.

The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.

3. Decide on a vocabulary of predicates, functions, and constants. That is, translate the important domain-level concepts into logic-level names.

This involves many questions of knowledge-engineering *style*.

Like programming style, this can have a significant impact on the eventual success of the project.

- For example, should pits be represented by objects or by a unary predicate on squares?
- Should the agent's orientation be a function or a predicate?
- Should the wumpus's location depend on time?
- Once the choices have been made, the result is a vocabulary that is known as the **ontology** of the domain.
- The word *ontology* means a particular theory of the nature of being or existence.
- The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.

4. Encode general knowledge about the domain.

- The knowledge engineer writes down the axioms for all the vocabulary terms.
- This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content.
- Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.

5. Encode a description of the specific problem instance.

- If the ontology is well thought out, this step will be easy.
- It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology.
- For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.

6. Pose queries to the inference procedure and get answers.

This is where the reward is:

- we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing.

- Thus, we avoid the need for writing an application-specific solution algorithm.

7. Debug the knowledge base.

The answers to queries will seldom be correct on the first try. More precisely, the answers will be correct *for the knowledge base as written,* assuming that the inference procedure is sound, but they will not be the ones that the user is expecting.

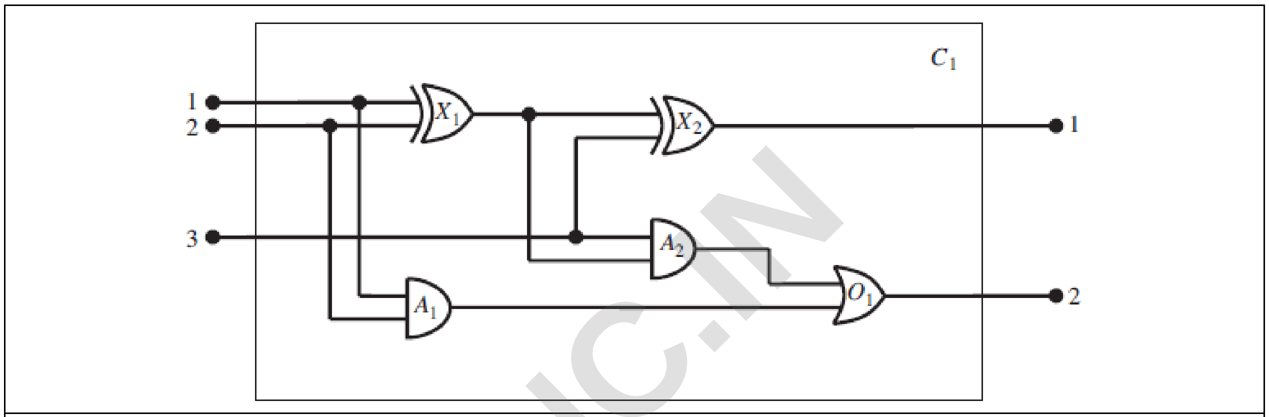


Figure 8.6 A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one OR gate.

- **Identify the TASK**
- There are many reasoning tasks associated with digital circuits.
- At the highest level, one analyzes the circuit's functionality. For example, does the circuit in Figure 8.6 actually add properly?
- If all the inputs are high, what is the output of gate A2?
- Questions about the circuit's structure are also interesting.
- For example, what are all the gates connected to the first input terminal?
- Does the circuit contain feedback loops?
- These will be our tasks in this section.
- There are more detailed levels of analysis, including those related to timing delays, circuit **area,** power consumption, production cost, and so on.
- Each of these levels would require additional knowledge.

Assemble the relevant knowledge

- What do we know about digital circuits? For our purposes, they are composed of wires and gates.
- Signals flow along wires to the input terminals of gates, and each gate produces a signal on the output terminal that flows along another wire.
- To determine what these signals will be, we need to know how the gates transform their input signals.
- There are four types of gates: AND, OR, and XOR gates have two input terminals, and NOT gates have one.
- All gates have one output terminal.
- Circuits, like gates, have input and output terminals.

Decide on Vocabulary

- The next step is to choose functions, predicates, and constants to represent them.
- First, we need to be able to distinguish gates from each other and from other objects.
- Each gate is represented as an object named by a constant, about which we assert that it is a gate with, say, Gate(X1).
- The behavior of each gate is determined by its type: one of the constants AND,OR, XOR, or NOT.
- Because a gate has exactly one type, a function is appropriate: Type(X1)=XOR.
- Circuits, like gates, are identified by a predicate: Circuit(C1).
- Next we consider terminals, which are identified by the predicate Terminal (x).
- A gate or circuit can have one or more input terminals and one or more output terminals.
- We use the function In(1,X1) to denote the first input terminal for gate X1.
- A similar function Out is used for output terminals.
- **The function Arity(c, i, j) says that circuit c has i input and j output terminals**.
- The connectivity between gates can be represented by a predicate, Connected, which takes two terminals as arguments, as in Connected(Out(1,X1), In(1,X2)).
- Finally, we need to know whether a signal is on or off.
- One possibility is to use a unary predicate, On(t), which is true when the signal at a terminal is on.
- This makes it a little difficult, however, to pose questions such as "What are all the possible values of the signals at the output terminals of circuit C1 ?"
- We therefore introduce as objects two signal values, 1 and 0, and a function Signal (t) that denotes

the signal value for the terminal t.

- If two terminals are connected, then they have the same signal:

∀ t1, t2 Terminal (t1) ∧ Terminal (t2) ∧ Connected(t1, t2) ⇒ Signal (t1)=Signal (t2) .

2. The signal at every terminal is either 1 or 0:
   $\forall t \;\; Terminal(t) \;\Rightarrow\; Signal(t) = 1 \lor Signal(t) = 0 \;.$
3. Connected is commutative:
   $\forall t_1, t_2 \;\; Connected(t_1, t_2) \;\Leftrightarrow\; Connected(t_2, t_1) \;.$
4. There are four types of gates:
   $\forall g \;\; Gate(g) \land k = Type(g) \;\Rightarrow\; k = AND \lor k = OR \lor k = XOR \lor k = NOT \;.$
5. An AND gate's output is 0 if and only if any of its inputs is 0:
   $\forall g \;\; Gate(g) \land Type(g) = AND \;\Rightarrow$
   $Signal(Out(1, g)) = 0 \;\Leftrightarrow\; \exists n \;\; Signal(In(n, g)) = 0 \;.$
6. An OR gate's output is 1 if and only if any of its inputs is 1:
   $\forall g \;\; Gate(g) \land Type(g) = OR \;\Rightarrow$
   $Signal(Out(1, g)) = 1 \;\Leftrightarrow\; \exists n \;\; Signal(In(n, g)) = 1 \;.$
7. An XOR gate's output is 1 if and only if its inputs are different:
   $\forall g \;\; Gate(g) \land Type(g) = XOR \;\Rightarrow$
   $Signal(Out(1, g)) = 1 \;\Leftrightarrow\; Signal(In(1, g)) \neq Signal(In(2, g)) \;.$
8. A NOT gate's output is different from its input:
   $\forall g \;\; Gate(g) \land (Type(g) = NOT) \;\Rightarrow$
   $Signal(Out(1, g)) \neq Signal(In(1, g)) \;.$
9. The gates (except for NOT) have two inputs and one output.
   $\forall g \;\; Gate(g) \land Type(g) = NOT \;\Rightarrow\; Arity(g, 1, 1) \;.$
   $\forall g \;\; Gate(g) \land k = Type(g) \land (k = AND \lor k = OR \lor k = XOR) \;\Rightarrow$
   $Arity(g, 2, 1)$
10. A circuit has terminals, up to its input and output arity, and nothing beyond its arity:
   $\forall c, i, j \;\; Circuit(c) \land Arity(c, i, j) \;\Rightarrow$
   $\forall n \;\; (n \leq i \;\Rightarrow\; Terminal(In(c, n))) \land (n > i \;\Rightarrow\; In(c, n) = Nothing) \land$
   $\forall n \;\; (n \leq j \;\Rightarrow\; Terminal(Out(c, n))) \land (n > j \;\Rightarrow\; Out(c, n) = Nothing)$
11. Gates, terminals, signals, gate types, and *Nothing* are all distinct.
   $\forall g, t \;\; Gate(g) \land Terminal(t) \;\Rightarrow$
   $g \neq t \neq 1 \neq 0 \neq OR \neq AND \neq XOR \neq NOT \neq Nothing \;.$
12. Gates are circuits.
   $\forall g \;\; Gate(g) \;\Rightarrow\; Circuit(g)$

Encode the specific problem instance

- First, we categorize the circuit and its component gates:
- Circuit(C1) ∧ Arity(C1, 3, 2)
- Gate(X1) ∧ Type(X1)=XOR
- Gate(X2) ∧ Type(X2)=XOR

- Gate(A1) ∧ Type(A1)=AND
- Gate(A2) ∧ Type(A2)=AND
- Gate(O1) ∧ Type(O1)=OR .
- **X1** (XOR): Computes **A ⊕ B**.
- **X2** (XOR): Computes **(A ⊕ B) ⊕ C_in** to produce the **Sum** output.
- **A1** (AND): Computes **A ∧ B**.
- **A2** (AND): Computes **C_in ∧ (A ⊕ B)**.
- **O1** (OR): Computes the **Carry-out** (C_out) as the OR of the two AND gates' outputs.
- Then, we show the connections between them:
- Connected(Out(1,X1), In(1,X2)) Connected(In(1,C1), In(1,X1))
- Connected(Out(1,X1), In(2,A2)) Connected(In(1,C1), In(1,A1))
- Connected(Out(1,A2), In(1,O1)) Connected(In(2,C1), In(2,X1))
- Connected(Out(1,A1), In(2,O1)) Connected(In(2,C1), In(2,A1))
- Connected(Out(1,X2), Out(1,C1)) Connected(In(3,C1), In(2,X2))
- Connected(Out(1,O1), Out(2,C1)) Connected(In(3,C1), In(1,A2)) .
- **1. Connected(Out(1,X1), In(1,X2))**
- **Out(1, X1)**: The output of the first XOR gate **X1**.
- **In(1, X2)**: The first input of the second XOR gate **X2**.
- This statement means that the output of **X1** is connected to the first input of **X2**. In the full adder, **X1** calculates **A ⊕ B**, and this output is used as an input to **X2**, which calculates **(A ⊕ B) ⊕ C_in** to produce the **Sum**.
- **2. Connected(In(1,C1), In(1,X1))**
- **In(1, C1)**: The first input of the circuit **C1** (which has 3 inputs and 2 outputs, representing the 1-bit full adder).
- **In(1, X1)**: The first input of XOR gate **X1**.
- This connection means that **A** (from the inputs of the full adder) is connected to the first input of XOR gate **X1**.
- **3. Connected(Out(1, X1), In(2, A2))**
- **Out(1, X1)**: The output of XOR gate **X1**.
- **In(2, A2)**: The second input of AND gate **A2**.
- This means the output of **X1** (which is **A ⊕ B**) is connected to the second input of the AND gate

A2, where it is used as **(A ⊕ B)** in the carry calculation.

- **4. Connected(In(1, C1), In(1, A1))**
- **In(1, C1)**: The first input of the carry input **C1**.
- **In(1, A1)**: The first input of AND gate **A1**.
- This means **C_in** (the carry input) is connected to the first input of **A1**, which calculates **A ∧ B**.
- **5. Connected(Out(1, A2), In(1, O1))**
- **Out(1, A2)**: The output of AND gate **A2**.
- **In(1, O1)**: The first input of OR gate **O1**.
- The output of **A2** (which is **C_in ∧ (A ⊕ B)**) is connected to the first input of OR gate **O1**, which is part of the final carry-out calculation.
- **6. Connected(In(2, C1), In(2, X1))**
- **In(2, C1)**: The second input of the carry input **C1**.
- **In(2, X1)**: The second input of XOR gate **X1**.
- This indicates that **B** is connected to the second input of **X1** (to calculate **A ⊕ B**).
- **7. Connected(Out(1, A1), In(2, O1))**
- **Out(1, A1)**: The output of AND gate **A1**.
- **In(2, O1)**: The second input of OR gate **O1**.
- The output of **A1** (which is **A ∧ B**) is connected to the second input of OR gate **O1**, which combines the results of the AND gates to calculate **Carry-out**.
- **8. Connected(In(2, C1), In(2, A1))**
- **In(2, C1)**: The second input of the carry input **C1**.
- **In(2, A1)**: The second input of AND gate **A1**.
- This connection suggests that **C_in** (the carry input) is connected to the second input of AND gate **A1**, participating in the calculation of **Carry-out** through the AND gates.
- **9. Connected(Out(1, X2), Out(1, C1))**
- **Out(1, X2)**: The output of XOR gate **X2** (which is the final **Sum** output).
- **Out(1, C1)**: The output of the full adder circuit **C1** (the final result of the adder).
- This means that the **Sum** output of the full adder (**Out(1, X2)**) is connected to the final output of the circuit **C1**.
- **10. Connected(In(3, C1), In(2, X2))**
- **In(3, C1)**: The third input of the full adder circuit **C1**.
- **In(2, X2)**: The second input of XOR gate **X2**.

- This means that **C_in** is connected to the second input of **X2**, as it is used to compute **(A ⊕ B) ⊕ C_in** for the **Sum** output.
- **11. Connected(Out(1, O1), Out(2, C1))**
- **Out(1, O1)**: The output of OR gate **O1** (the **Carry-out**).
- **Out(2, C1)**: The second output of the full adder circuit **C1**.
- This means that the **Carry-out** from **O1** is connected to the **Carry-out** output of the full adder circuit.
- **12. Connected(In(3, C1), In(1, A2))**
- **In(3, C1)**: The third input of the full adder circuit **C1**.
- **In(1, A2)**: The first input of AND gate **A2**.
- This means that **C_in** is connected to the first input of AND gate **A2**, which is used in the calculation of **C_in ∧ (A ⊕ B)** for the carry computation.

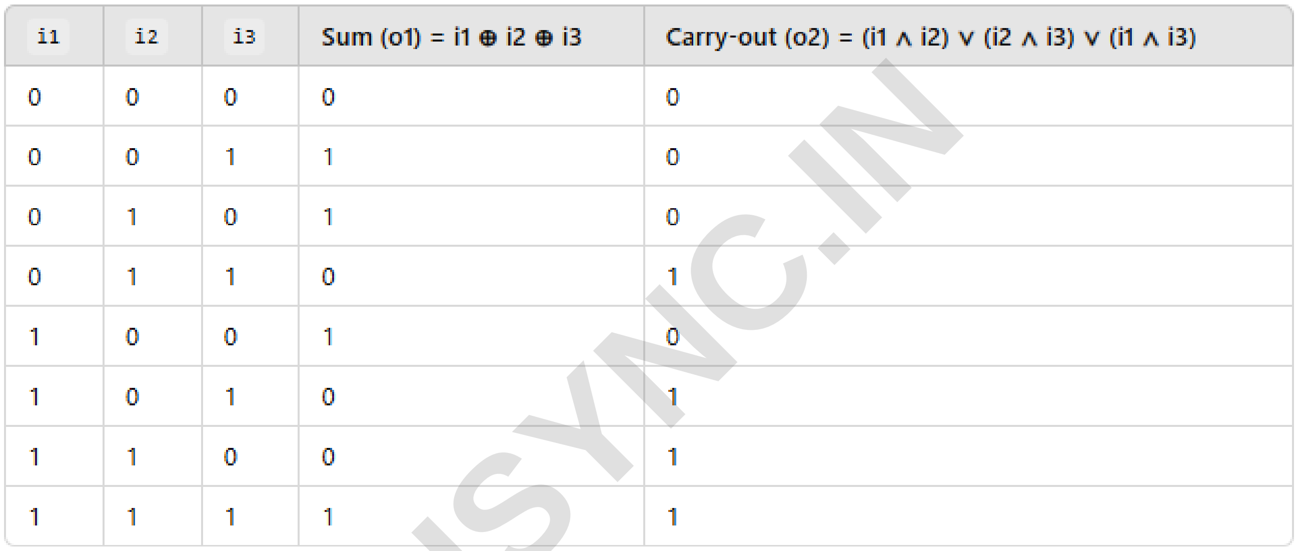
Summary:

- The **1-bit full adder circuit** involves three types of gates: **XOR**, **AND**, and **OR**.
- The inputs are **A**, **B**, and **C_in** (carry input), and the outputs are **Sum** and **Carry-out**.
- The various **Connected** statements describe how these gates interact with each other, ensuring the proper calculation of the **Sum** and **Carry-out**.
- The **Sum** is calculated by XORing **A**, **B**, and **C_in**, and the **Carry-out** is calculated using AND and OR gates, based on the inputs.
- The connections reflect the standard logic for a **1-bit full adder** where:
- **Sum** = **(A ⊕ B) ⊕ C_in**.
- **Carry-out** = **(A ∧ B) ∨ (C_in ∧ (A ⊕ B))**.

Pose queries to the inference procedure

- What combinations of inputs would cause the first output of C1 (the sum bit) to be 0 and the second output of C1 (the carry bit) to be 1?
- ∃ i1, i2, i3 Signal (In(1, C1))=i1 ∧ Signal (In(2, C1))=i2 ∧ Signal (In(3, C1))=i3∧ Signal (Out(1, C1))=0 ∧ Signal (Out(2, C1))=1 .
- The answers are substitutions for the variables i1, i2, and i3 such that the resulting sentence is entailed by the knowledge base.
- ASKVARS will give us three such substitutions:
- {i1/1, i2/1, i3/0} {i1/1, i2/0, i3/1} {i1/0, i2/1, i3/1} .

- What are the possible sets of values of all the terminals for the adder circuit?
- ∃ i1, i2, i3, o1, o2 Signal (In(1, C1))=i1 ∧ Signal (In(2, C1))=i2
- ∧ Signal (In(3, C1))=i3 ∧ Signal (Out(1, C1))=o1 ∧ Signal (Out(2, C1))=o2 .
- {i1/1, i2/1, i3/0}: Sum = 1 ⨁ 1 ⨁ 0 = 0 (sum is 0) and carry-out = (1 ∧ 1) ∨ (1 ∧ 0) ∨ (1 ∧ 0) = 1 (carry-out is 1).
- Cout=(A∧B)∨(B∧Cin)∨(A∧Cin)



i1	i2	i3	Sum (o1) = i1 ⊕ i2 ⊕ i3	Carry-out (o2) = (i1 ∧ i2) ∨ (i2 ∧ i3) ∨ (i1 ∧ i3)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

For example:

{i1=0, i2=0, i3=0, o1=0, o2=0}

Debug the knowledge base

- Various ways to see what kinds of erroneous behaviors emerge.
- For example, suppose we fail to read alternative semantics and hence forget to assert that 1 ≠ 0.
- We can pinpoint the problem by asking for the outputs of each gate. For example, we can ask
- ∃ i1, i2, o Signal (In(1,C1))=i1 ∧ Signal (In(2,C1))=i2 ∧ Signal (Out(1,X1)) ,
- which reveals that no outputs are known at X1 for the input cases 10 and 01.
- Then, we look at the axiom for XOR gates, as applied to X1:
- Signal (Out(1,X1))=1 ⇔ Signal (In(1,X1)) = Signal (In(2,X1)) .
- If the inputs are known to be, say, 1 and 0, then this reduces to
- Signal (Out(1,X1))=1 ⇔ 1 ≠ 0 .
- Now the problem is apparent: the system is unable to infer that Signal (Out(1,X1))=1, so we need to tell it that 1 ≠ 0.

- The system is unable to infer the correct behavior of the XOR gate for certain input combinations, like 1 and 0.
- This failure to infer is due to the lack of knowledge about the relationship between those inputs (i.e., that 1 and 0 are different).
- By examining the axiom for the XOR gate and testing for the output at each gate, it becomes clear that the system needs to be explicitly told about the condition 1 ≠ 0 in order to deduce the correct output.
- Once this information is provided, the system can correctly infer that Signal (Out(1, X1)) = 1 when the inputs are 1 and 0.

In essence, the problem is a missing or forgotten **assertion** that would allow the system to properly deduce the XOR gate's output.

INFERENCE IN FIRST ORDER LOGIC

- Propositional vs First order inference
- Inference Rules for Quantifiers

All greedy kings are evil

- ∀ x King(x) ∧ Greedy(x) ⇒ Evil(x) .
- Then it seems quite permissible to infer any of the following sentences:
- King(John) ∧ Greedy(John) ⇒ Evil(John)
- King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)
- King(Father (John)) ∧ Greedy(Father (John)) ⇒ Evil(Father (John)) .
- The rule of **Universal Instantiation** (**UI** for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable.

- Let SUBST(θ,α) denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v\ \alpha}{\text{SUBST}(\{v/g\}, \alpha)} \qquad \text{for any variable } v \text{ and ground term } g.$$

- For example, the three sentences given earlier are obtained with the substitutions {x/John}, {x/Richard }, and {x/Father (John)}.
- In the rule for **Existential Instantiation**, the variable is replaced by a single *new constant symbol*. The formal statement is as follows: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$$\frac{\exists v\ \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

- For example, from the sentence
- ∃ x Crown(x) ∧ OnHead(x, John)
- we can infer the sentence
- Crown(C1) ∧ OnHead(C1, John)
- as long as C1 does not appear elsewhere in the knowledge base.
- Basically, the existential sentence says there is some object satisfying a condition, and applying the existential instantiation rule just gives a name to that object. Of course, that name must not already belong to another object.

Reduction to propositional Inference

- Let's consider an example:
- **PredicateLogicStatement:**

  ∀x (Person(x) → ∃y (FriendOf(y, x)))
- This statement says: "For all x, if x is a person, there exists a y such that y is a friend of x."

Step 1: Eliminate quantifiers by replacing them with specific constants.

- "Person(x)" becomes propositional variables like Person(a), Person(b).
- "FriendOf(y, x)" becomes FriendOf(a, b), FriendOf(b, a), and so on.

Step 2: Convert the logical operations (→ and ∃) into simpler logical connectives.

- "Person(x) → ∃y (FriendOf(y, x))" becomes "¬Person(x) ∨ ∃y FriendOf(y, x)"
- (implication is replaced by disjunction and negation).

Step 3: Replace existential quantifiers with propositional variables, e.g., FriendOf(a, b).

- Thus, the original statement can be reduced to a set of propositional logic clauses.

UNIFICATION AND LIFTING

- Unification is the process of finding a substitution for variables in two logical expressions so that the expressions become identical. In essence, the goal is to make two formulas "look the same" by replacing variables with terms (constants or functions).

- A first order Inference rue:**Generalized Modus Ponen's**
- For atomic sentences pi, pi′, and q, where there is a substitution
- θ

$$\text{such that } \mathrm{SUBST}(\theta, p_i{}') = \mathrm{SUBST}(\theta, p_i), \text{ for all } i,$$

$$\frac{p_1{}',\ p_2{}',\ \ldots,\ p_n{}',\ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\mathrm{SUBST}(\theta, q)}\ .$$

Generalized Modus Ponens is a **lifted** version of Modus Ponens—it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic.

Premises

We have a series of **n atomic sentences** (propositions), denoted by:

- $p_1, p_2, \ldots, p_n$
- $P_1$is: King(x)
- p_2 is: Greedy(x)

Additionally, there is one **implication** involving these premises:

- $(p_1 \wedge p_2 \wedge \cdots \wedge p_{2n} \Rightarrow q)$
- This implication states that if all the premises $p_1, p_2, \ldots, p_n$ are true, then the conclusion q must also be true.
- $(P_1 \wedge P_2 \Rightarrow q)$ where q is: Evil(x)

So, the implication says:

- "If P_1 (King(x)) and P_1 (Greedy(x)) are true, then q(Evil(x)) must also be true."

Substitution (θ)

The substitution θ is given as:

- θ={x/John,y/John}

This substitution says:

- Replace x with **John** in all occurrences of x.
- Replace y with **John** in all occurrences of y.

Thus, when you apply this substitution to a formula, you replace the variables x and y with the constant "John."

Substitution Applied to the Conclusion (q)

The final part of the rule is applying the substitution θ to the consequent q, which is Evil(x) q is Evil(x) meaning "x is evil."

- After applying the substitution θ={x/John,y/John}, you replace the variable x with "John".

Therefore:

- SUBST(θ,q) becomes Evil(John)

So, after substitution, the conclusion becomes Evil(John), which asserts that John is evil.

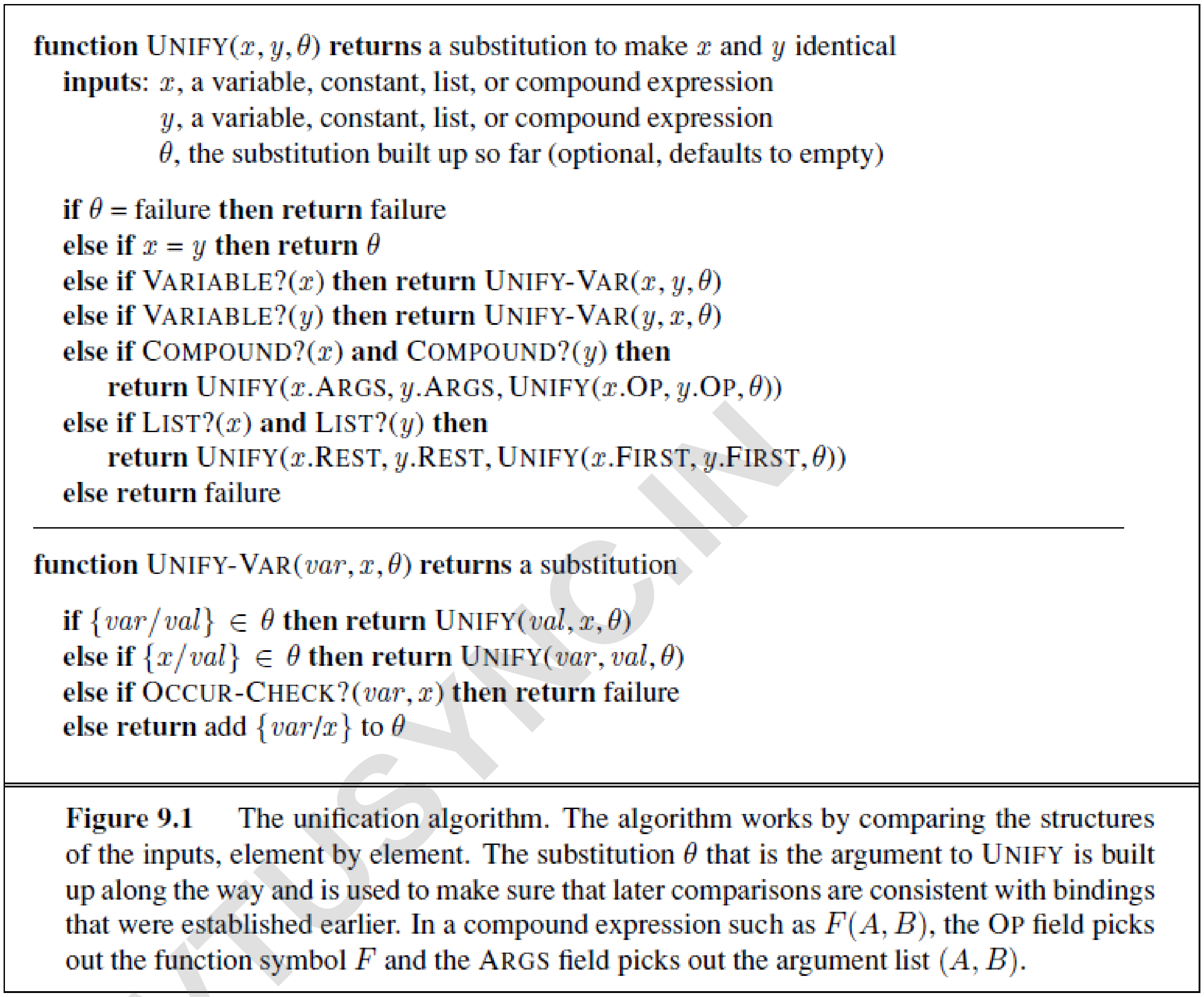
Here’s a complete breakdown of how this rule works in the context of the given example:

1. **Premises**:
    1. p1:King(x) (John is a King)
    2. p2:Greedy (John is Greedy)
    3. The rule: If p1and p2 are true, then q (Evil(x)) is true.
2. **Implication (Rule):**
    1. (p1∧p2⇒q) which reads: "If John is a King and greedy, then John is evil."
3. **Substitution**:
    1. θ={x/John,y/John}, meaning replace all instances of x and y with "John."
4. **Conclusion after applying the substitution**:
    1. q=Evil(x), and after applying θ , it becomes Evil(John).

Thus, the result of applying the substitution θ to q is that John is evil.

UNIFICATION

- UNIFY(Knows(John, x), Knows(John, Jane)) = {x/Jane}
    - Substitution: x→Jane
- UNIFY(Knows(John, x), Knows(y, Bill)) = {x/Bill, y/John}
    - Substitution: x→Bill,y→John
- UNIFY(Knows(John, x), Knows(y,Mother (y))) = {y/John, x/Mother (John)}
    - Substitution: y→John,x→Mother(John)
- UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail .
    - **Fail**: The unification fails because of a contradiction in the substitution for x.



```
function UNIFY($x, y, \theta$) returns a substitution to make $x$ and $y$ identical
  inputs: $x$, a variable, constant, list, or compound expression
          $y$, a variable, constant, list, or compound expression
          $\theta$, the substitution built up so far (optional, defaults to empty)

  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($x$.ARGS, $y$.ARGS, UNIFY($x$.OP, $y$.OP, $\theta$))
  else if LIST?($x$) and LIST?($y$) then
     return UNIFY($x$.REST, $y$.REST, UNIFY($x$.FIRST, $y$.FIRST, $\theta$))
  else return failure
```

```
function UNIFY-VAR($var, x, \theta$) returns a substitution

  if $\{var/val\} \in \theta$ then return UNIFY($val, x, \theta$)
  else if $\{x/val\} \in \theta$ then return UNIFY($var, val, \theta$)
  else if OCCUR-CHECK?($var, x$) then return failure
  else return add $\{var/x\}$ to $\theta$
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

•

Example 1:

We want to unify the terms

Knows(John, x) and Knows(John, Jane).

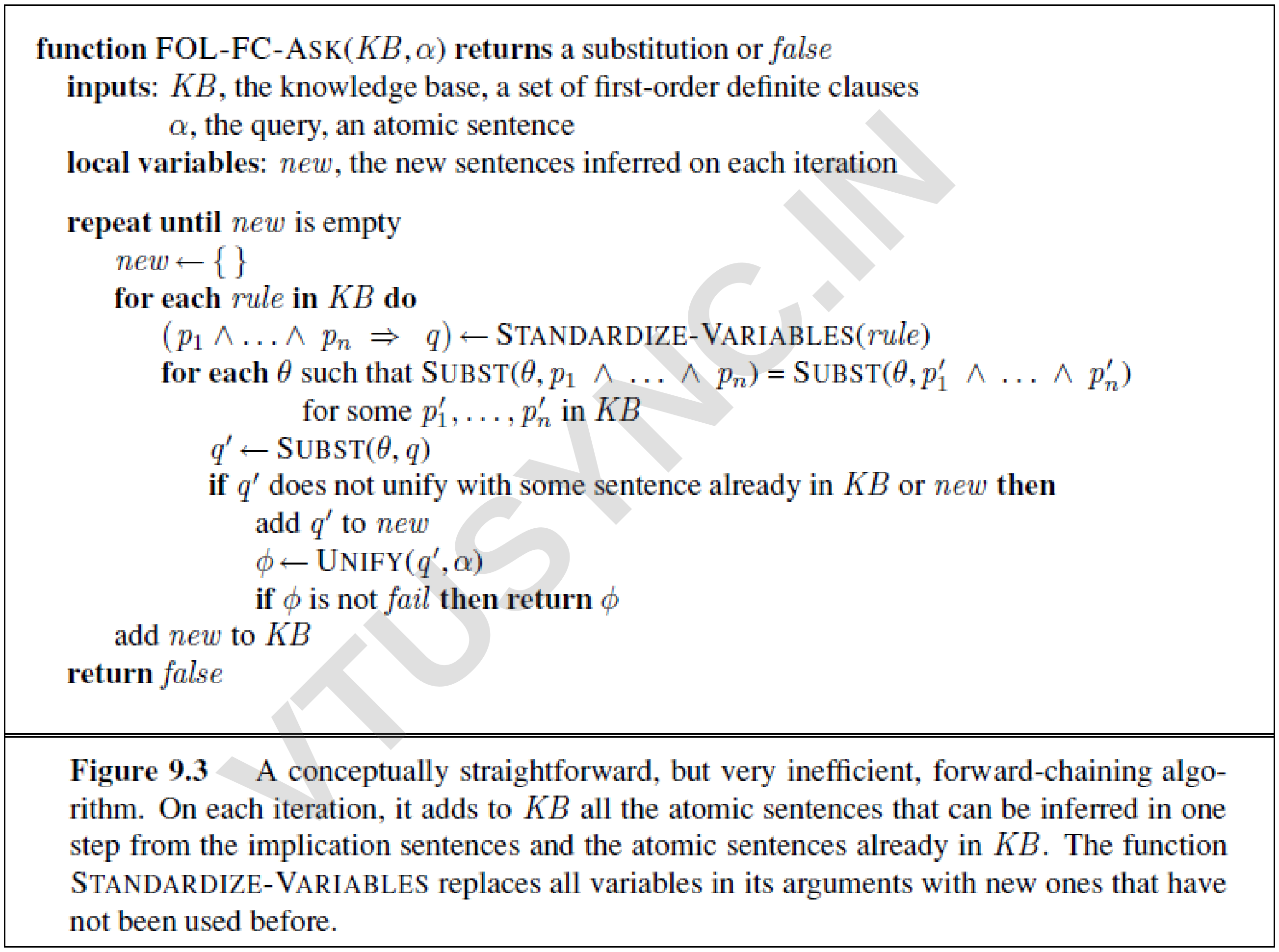
- The operator Knows is the same for both terms.
- The first argument is John in both cases, so they are already unified.
- The second argument is x in the first term and Jane in the second term. We need to unify x with Jane.

The unification is possible, and the substitution becomes {x → Jane}.

FORWARD CHAINING

- The idea is simple: start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made.

- Here, we explain how the algorithm is applied to first-order definite clauses.
- Definite clauses such as Situation ⇒ Response are especially useful for systems that make inferences in response to newly arrived information.
- Forward Chaining is a method of reasoning used in rule-based systems and expert systems. It is a form of data-driven reasoning where the inference engine starts with the available facts and applies inference rules to extract new facts, progressively working forward through the knowledge base.



```
function FOL-FC-ASK($KB, \alpha$) returns a substitution or $false$
   inputs: $KB$, the knowledge base, a set of first-order definite clauses
           $\alpha$, the query, an atomic sentence
   local variables: $new$, the new sentences inferred on each iteration

   repeat until $new$ is empty
       $new \leftarrow \{ \}$
       for each $rule$ in $KB$ do
           $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES($rule$)
           for each $\theta$ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
                       for some $p'_1, \ldots, p'_n$ in $KB$
               $q' \leftarrow$ SUBST($\theta, q$)
               if $q'$ does not unify with some sentence already in $KB$ or $new$ then
                   add $q'$ to $new$
                   $\phi \leftarrow$ UNIFY($q', \alpha$)
                   if $\phi$ is not $fail$ then return $\phi$
       add $new$ to $KB$
   return $false$
```

Figure 9.3 A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

Eg:

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by ColonelWest, who is American.

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

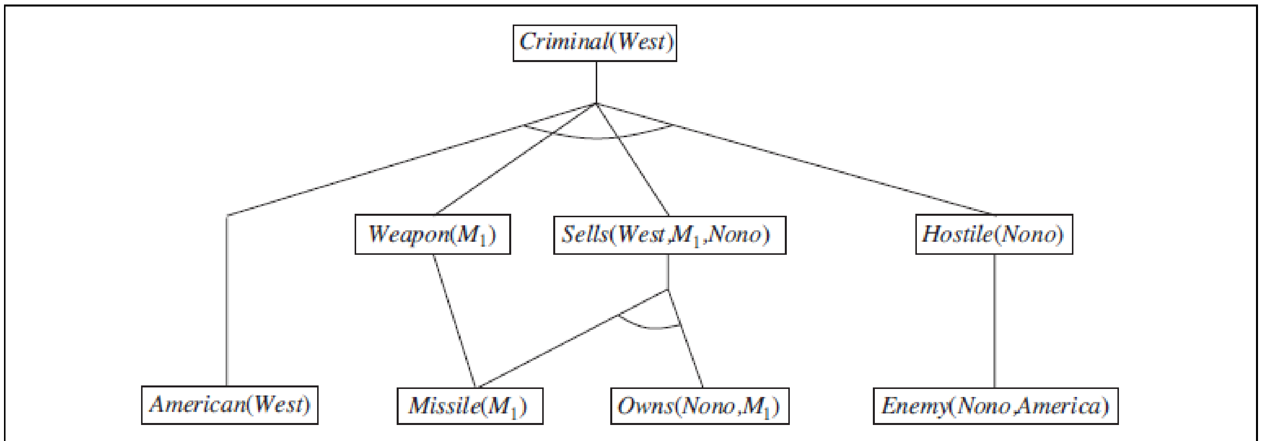


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

Algorithm Steps Explained:

1. Initialize:

- The algorithm starts by initializing the **new** set to empty. This set will collect all newly inferred facts on each iteration.

2. Repeat Until New Facts Are Inferred:

- The loop continues until no new facts are inferred, i.e., when the **new** set is empty. This ensures that the algorithm will keep applying the rules to infer facts until no further progress can be made.

3. Standardizing Variables in Rules:

- For each rule in the knowledge base, the algorithm applies **standardization**. The function **STANDARDIZE-VARIABLES(rule)** ensures that all variables in the rule are renamed so that no two rules share the same variable names. This prevents variable conflicts during unification.
  - Example: If two rules have variables x and y, **STANDARDIZE-VARIABLES** will rename them (e.g., x_1 and y_1) to avoid confusion when performing unification.

4. Try to Match Rules with Known Facts:

- For each rule in the KB, it looks for **substitutions θ** that match the left-hand side (the **premises**) of the rule with some existing facts in the KB.
- It checks all combinations of the premises p_1 ,p_2 ,…, p_n from the KB (i.e., the left side of the rule).
- The algorithm compares the premises of the rule against the existing facts (which could be in KB or **new** facts) to see if any of them unify with the premises. If a match is found, the rule's conclusion q is derived using the unification of those facts.

5. Apply Substitution to the Conclusion:

- If a match is found (i.e., a valid substitution θ is found), the algorithm applies the substitution to the conclusion q of the rule:
    - **q' = SUBST(θ, q)**: This means the right-hand side (conclusion) of the rule is modified using the substitution θ.

6. Check for Redundant Conclusions:

- After obtaining q′, the algorithm checks if q′already exists in the KB or in the set of newly inferred facts (**new**).
    - If q′is already present, it is not added again.
    - If q′ is new, it is added to the **new** set. This ensures that only new, unique facts are added to the knowledge base.

7. Unify with the Query:

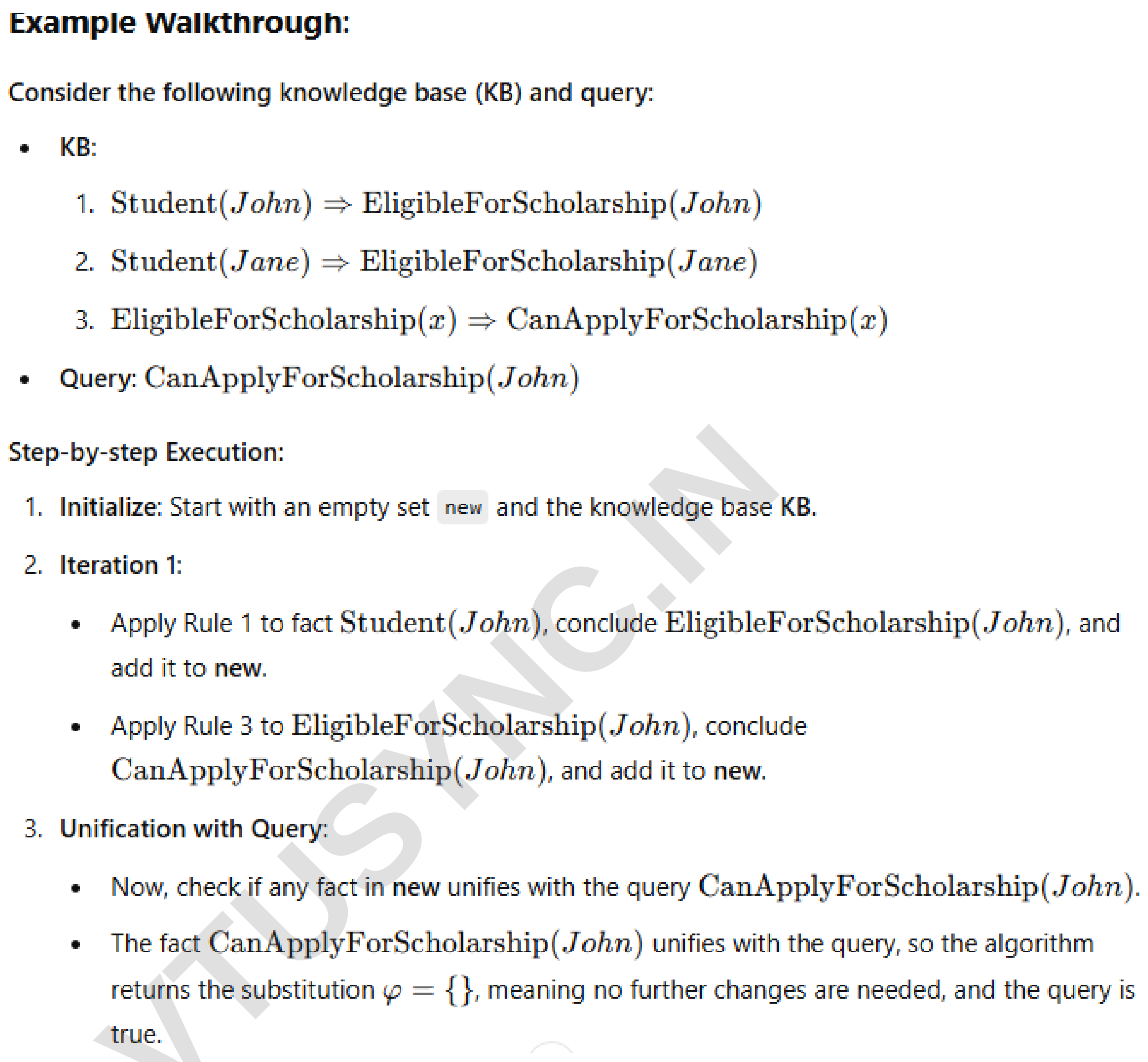
- After the fact q′ has been added to the **new** set, the algorithm checks if this fact unifies with the query α.
    - **φ = UNIFY(q', α)**: The function **UNIFY** tries to find a substitution φ that makes q′and α identical.
    - If **UNIFY** succeeds (i.e., returns a non-failing substitution φ), then the query is provable from the knowledge base, and the algorithm returns φ, the substitution that makes the query true.

8. Add New Facts to KB:

- If no valid unification is found for any q′, the **new** facts are added to the knowledge base **KB**.
    - This is done so that these newly inferred facts can be used in the next iteration to infer further facts.

9. Return False if Query is Not Provable:

- If, after all iterations, no unification is found for α, meaning no substitution makes the query true, the algorithm returns **false**. This means that the query α cannot be derived from the knowledge base.



Example Walkthrough:

Consider the following knowledge base (KB) and query:

- KB:
  1. $\text{Student}(John) \Rightarrow \text{EligibleForScholarship}(John)$
  2. $\text{Student}(Jane) \Rightarrow \text{EligibleForScholarship}(Jane)$
  3. $\text{EligibleForScholarship}(x) \Rightarrow \text{CanApplyForScholarship}(x)$
- Query: $\text{CanApplyForScholarship}(John)$

Step-by-step Execution:

1. **Initialize:** Start with an empty set `new` and the knowledge base **KB**.
2. **Iteration 1:**
   - Apply Rule 1 to fact $\text{Student}(John)$, conclude $\text{EligibleForScholarship}(John)$, and add it to **new**.
   - Apply Rule 3 to $\text{EligibleForScholarship}(John)$, conclude $\text{CanApplyForScholarship}(John)$, and add it to **new**.
3. **Unification with Query:**
   - Now, check if any fact in **new** unifies with the query $\text{CanApplyForScholarship}(John)$.
   - The fact $\text{CanApplyForScholarship}(John)$ unifies with the query, so the algorithm returns the substitution $\varphi = \{\}$, meaning no further changes are needed, and the query is true.

- The idea is simple: start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made.
- This method uses First Order Definite clause .
- A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal.

For example

- King(x) ∧ Greedy(x) ⇒ Evil(x) .
- King(John) .
- Greedy(y) .
- The law says that it is a crime for an American to sell weapons to hostile nations. The country

Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by ColonelWest, who is American.

- We will prove that West is a criminal.
- Representation of this by using first order definite clauses.
- So it is a crime for an American to sell weapons to hostile nations.
- American(x) ∧Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z) ⇒ Criminal (x) ----1
- "Nono . . . has some missiles."
- The sentence ∃ x Owns(Nono, x)∧Missile(x) is transformed into two definite clauses by Existential Instantiation, introducing a new constant M1:
- Owns(Nono,M1) ---2
- Missile(M1)-----3
- "All of its missiles were sold to it by Colonel West":

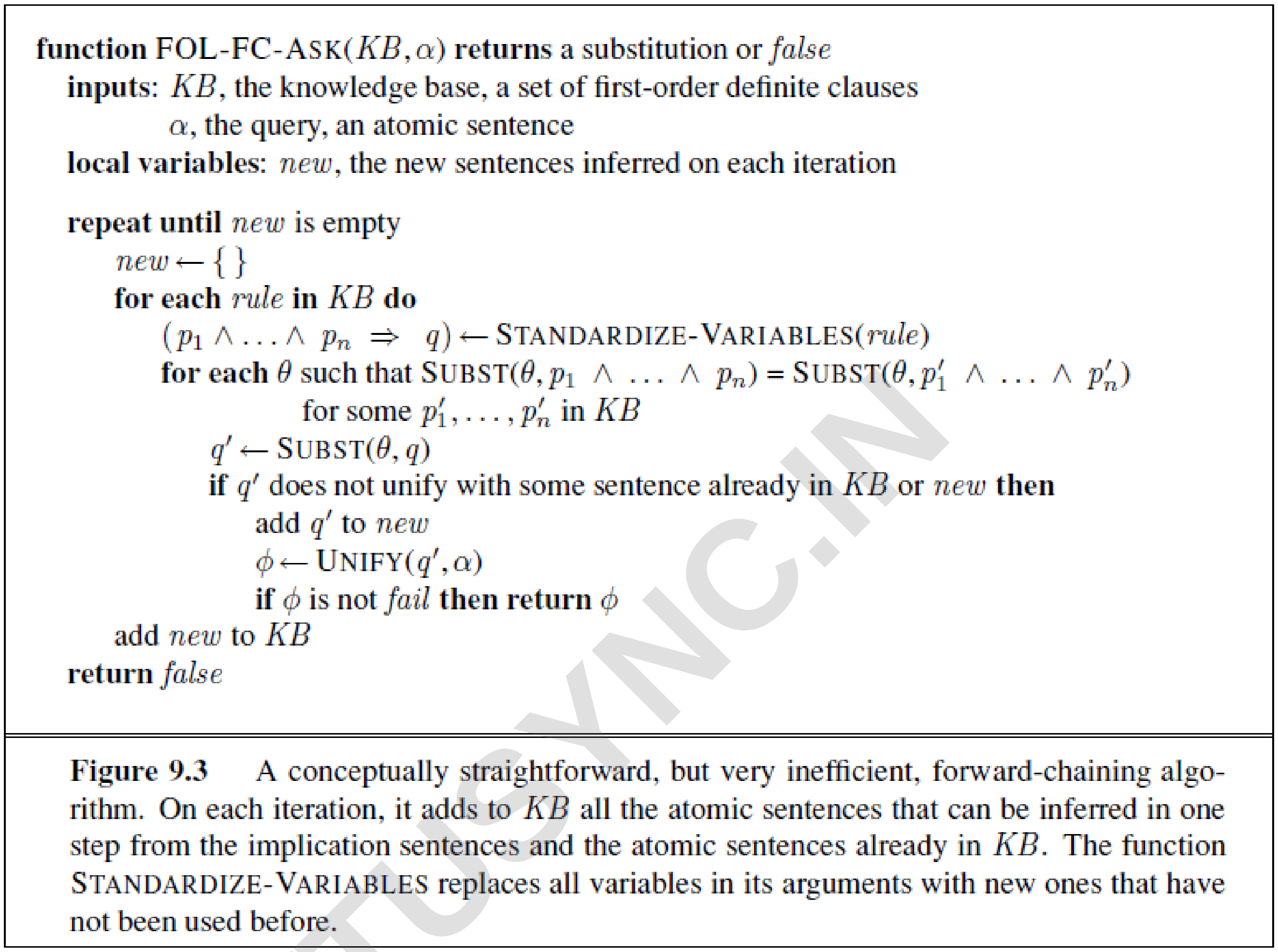
  Missile(x) ∧ Owns(Nono, x) ⇒ Sells(West, x, Nono) ---4
- We will also need to know that missiles are weapons:

  Missile(x) ⇒ Weapon(x) ----5
- An enemy of America counts as "hostile":
- Enemy(x,America) ⇒ Hostile(x) ----6
- "West, who is American . . .":
- American(West) -----7
- "The country Nono, an enemy of America . . .":
- Enemy(Nono,America) .
- This knowledge base contains no function symbols and is therefore an instance of the class

of Datalog knowledge bases.

- Datalog is a language that is restricted to first-order definite clauses with no function symbols.
- Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered or no new facts are added.
- On the first iteration, rule 1 has unsatisfied premises.
- Rule 4 is satisfied with {x/M1}, and Sells(West,M1, Nono) is added.
- Rule 5 is satisfied with {x/M1}, and Weapon(M1) is added.
- Rule 6 is satisfied with {x/Nono}, and Hostile(Nono) is added.
- On the second iteration, rule 1 is satisfied with {x/West, y/M1, z/Nono}, and Criminal (West) is

added.



```
function FOL-FC-ASK($KB$, $\alpha$) returns a substitution or false
  inputs: $KB$, the knowledge base, a set of first-order definite clauses
          $\alpha$, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
     new ← { }
     for each rule in $KB$ do
        $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(rule)
        for each $\theta$ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
                   for some $p'_1, \ldots, p'_n$ in $KB$
           $q' \leftarrow$ SUBST($\theta, q$)
           if $q'$ does not unify with some sentence already in $KB$ or new then
              add $q'$ to new
              $\phi \leftarrow$ UNIFY($q', \alpha$)
              if $\phi$ is not fail then return $\phi$
     add new to $KB$
  return false
```

Figure 9.3 A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

Steps:

1. Initialization
2. Loop until new is empty
3. Processing each rule
4. Finding applicable premises
5. Checking for duplicates
6. Unifying with the query
7. Adding new facts to KB
8. Termination

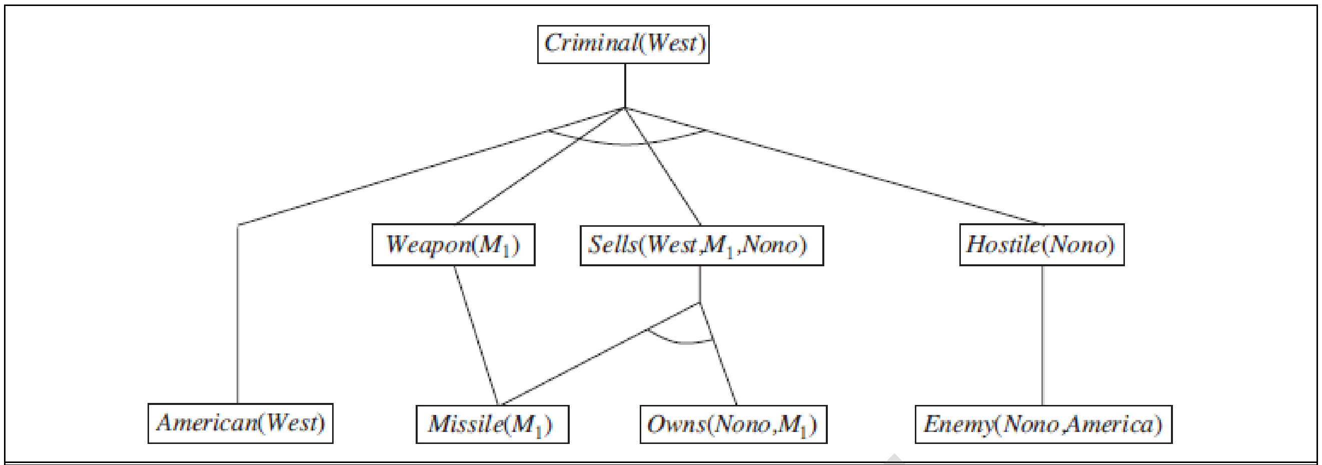


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

Efficient Forward Chaining

- There are three possible sources of inefficiency.
- First, the “inner loop” of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the knowledge base. This is often called **pattern matching** and can be very expensive.
- Second, the algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the knowledge base on each iteration.
- Finally, the algorithm might generate many facts that are irrelevant to the goal.