AI_Module4

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

Topics:

1. First Order Logic:
    a. **Representation Revisited,**
    b. **Syntax and Semantics of First Order Logic,**
    c. **Using First Order Logic**.
    d. **Knowledge Engineering in First Order Logic**
2. Inference in First Order Logic:
    a. **Propositional Versus First Order Inference,**
    b. **Unification,**
    c. **Forward Chaining,**

4.1 First Order Logic

4.1.a Representation of First Order Logic Revisited

This section explores *representation languages* and motivates the development of *first-order logic*, which is more expressive than propositional logic.

Programming Languages: Common programming languages (like **C++, Java, and Lisp**) are a major type of formal language. Programs represent computational processes and can use data structures to store facts.

- **Example**: A **4x4 array** can represent a game environment (like Wumpus World).
- **Statement**: `World[2,2]←Pit` means there is a pit at square [2,2].

Programming languages do not automatically derive new facts; updates depend on specific procedures created by the programmer.

Procedural vs. Declarative:

- **Procedural approach**: Updates to data structures require domain-specific knowledge and are not generalizable.
- **Declarative approach**: In propositional logic, knowledge and reasoning are separate and can apply universally across domains.

Limitations of Data Structures: It's hard to express complex statements in programming languages.

- **Example**: Saying "**There is a pit in [2,2] or [3,1**]" is not straightforward.

Programs can usually only store one value per variable and may not handle partial information well.

Compositionality: Propositional logic is **declarative** because it connects sentences with possible truths. It can express partial information using logical operations (like **disjunction** and **negation**).
Compositionality means the meaning of a statement depends on the meanings of its parts.

- **Example:** In "S1,4 ∧ S1,2," the meaning depends on "S1,4" and "S1,2."
- It's odd if different meanings are assigned to parts that do not relate to each other.

Need for First-Order Logic: Propositional logic struggles to describe complex environments.

- **Example:** In propositional logic, you must create separate rules for each square (like $B_{1,1} \Leftrightarrow (P_{1,2} \lor P_{2,1})$).
- In contrast, natural language can easily say, "**Squares adjacent to pits are breezy**," making it more efficient for concise descriptions.

Language Thought:

Natural languages, like English or Spanish, are very expressive. In the fields of linguistics and philosophy, natural language is often seen as a way to represent knowledge. If we could understand its rules completely, natural language could help us build systems that can reason and understand vast amounts of written information. Today, natural language is seen as more of a communication tool than a way to represent pure facts.

For example, if someone says, "Look!" we understand their intent by combining the word with the context (**like spotting Superman**). But on its own, the word "Look!" doesn't convey much information and needs context.

This **limitation** makes it hard to store natural language sentences in knowledge systems, as they would also need to store the context to fully capture meaning.

Another issue is **ambiguity;** some words have multiple meanings. As Pinker (1995) explained, people might think of the season "spring" or a metal coil, showing that words alone don't always capture clear meanings.

Sapir-Whorf Hypothesis and Language's Influence on Thought

- **Sapir-Whorf Hypothesis**: This hypothesis suggests that the language we speak **influences how we see and understand the world**. According to Whorf (1956), people use language to divide and categorize their experiences, based on shared understanding in their language community.
- **Language Shapes Perception**: Different languages categorize concepts differently. For example, French has two words, "**chaise**" and "**fauteuil**," for what English speakers call "chair." However, English speakers can still understand the idea of "**fauteuil**" (like an "open-arm chair"), questioning how much language shapes thinking. Whorf's ideas were based on observation, but studies since have used data from anthropology, psychology, and neuroscience to explore this theory.
- **Memory and Language**:
  - Experiment by Wanner (1974): Participants could recall the meaning **of sentences better than the exact words**. This suggests that people

process language in a nonverbal way, focusing more on meaning than specific words.

- **Language and Spatial Orientation**:
  - The Australian Aboriginal language **Guugu Yimithirr** doesn't use terms for "front," "back," "left," or "right." Instead, it uses absolute directions (north, south, etc.). This affects behavior: **Guugu Yimithirr** speakers are better at navigating using cardinal directions, while English speakers rely on relative directions.
- **Influence of Grammar and Vocabulary**:
  - Grammatical gender can influence perception. Example: The word "**bridge**" is masculine in **Spanish**, leading speakers to describe it as strong and towering, but feminine in **German**, where it's described as **elegant and fragile**.
  - Vocabulary can change perception too. In a study by Loftus and Palmer (1974), participants viewed a car accident and estimated higher speeds when the word "**smashed**" was used instead of "**contacted**," showing how word choice impacts perception.

Logic, Language, and Thought Representation

- **First-Order Logic and CNF**: In logic, forms like "¬(A ∨ B)" and "¬A ∧ ¬B" mean the same thing when stored in *Canonical Normal Form (CNF)*.CNF helps systems recognize these forms as identical, even if they look different on the surface.

- **Brain and Word Recognition with fMRI**: Until recently, it wasn't possible to know if the brain processed information in a similar way. However, studies by Mitchell et al. (2008) have shown that fMRI can identify patterns when people see certain words. By scanning brain images of people shown words like "celery" or "airplane," a computer could predict the word they were shown with 77% accuracy. Remarkably, this system even worked with words and people the program had never seen before, suggesting some common brain representation of words across people.

- **Importance of Representation Forms in Reasoning**: In logic, the specific way information is represented doesn't change the facts derived from it. But in practical reasoning, some representations are faster or more efficient. For tasks like learning from experience, the form of representation matters. Simpler, more concise representations can lead to faster conclusions and are often chosen by learning programs.

- **Language's Role in Learning and Decision-Making**: When a learning system encounters multiple valid explanations, it usually picks the simplest one.

This is known as the principle of choosing the most *succinct* theory. Since the language used impacts how theories are represented, language ultimately influences thought and learning processes.

Note: fMRI stands for **functional Magnetic Resonance Imaging**, a type of brain scan used to measure and map brain activity. Unlike regular MRI, which shows the structure of the brain, fMRI detects changes in blood flow to different areas of the brain, which increases in regions that are more active. Here's how it works:

1. **Blood Flow and Brain Activity**: When a specific brain region is active (e.g., during thinking, moving, or sensing), it requires more oxygen, which is carried by the blood. fMRI detects these changes in blood oxygen levels.
2. **Mapping Activity**: fMRI produces images showing which parts of the brain are working harder at any given moment. This allows researchers to see which areas are involved in various tasks like speaking, seeing, or remembering.
3. **Applications**: fMRI is widely used in research to understand how different parts of the brain function, to study brain disorders, and even to explore how people process language, memory, and emotions.

Since fMRI is non-invasive (it doesn't require surgery or radiation), it's a safe and popular tool for studying brain activity in real time.

Combining Formal and Natural Languages in Logic:

1. **Using Objects, Relations, and Functions**: We can create a logic system that's based on propositional logic (clear and context-independent) and add ideas from natural language. This allows us to express things more naturally but avoids ambiguity. For example:
   - **Objects**: Things like people, houses, numbers, colors, etc.
   - **Relations**: Connections between objects, like "brother of," "bigger than," or "has color."
   - **Functions**: Operations with a single output for each input, like "father of" or "one more than."

   Examples:

     - "One plus two equals three" uses objects (numbers), a relation (equals), and a function (plus).
     - "Squares neighboring the wumpus are smelly" has objects (squares, wumpus), a property (smelly), and a relation (neighboring).

2. **Ontological and Epistemological Commitments in Logic**:

Ontological Commitment: What a logic system assumes exists in the world.

- **Propositional logic**: Assumes simple facts are either true or false.
- **First-order logic**: Assumes objects and relations exist in the world.
- **Temporal logic**: Adds the idea of time.
- **Higher-order logic**: Treats relations and functions as objects.

Epistemological Commitment: What a logic system allows you to believe about these facts.

- **Propositional and first-order logic**: Allows true/false/unknown beliefs.
- **Probability and fuzzy logic**: Allows degrees of belief, from 0 to 1 (like a 75% chance).

The ontological and epistemological commitments of five different logics are summarized in the following Figure:

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

Fig: Formal languages and their ontological and epistemological commitments.

Properties of Propositional Logic

1. Propositional logic is a **declarative language** because its semantics is based on a truth relation between sentences and possible worlds.
2. It also has sufficient **expressive power to deal with partial information**, using disjunction and negation.
3. Propositional logic has a third property that is desirable in representation languages, namely, **compositionality.** In a compositional language, the meaning of a sentence is a function of the meaning of its parts**.**

For example, the meaning of "S1,4 ∧ S1,2" is related to the meanings of "S1,4" and "S1,2."

Drawbacks of Propositional Logic

- **Propositional logic** (PL) is declarative and assumes the world contains facts, so it guides us on how to represent information *in a logical form* and draw conclusions.
- We can only represent information as either **true or false** in propositional logic.
- Expressive power of Propositional logic is very limited and lacks to describe an environment with many objects
- If you want to represent complicated sentences or natural language statements, PL is not sufficient.
- **Examples:** PL is not enough to represent the sentences below, so we require powerful logic (such as FOL).
  1. I love mankind. It's the people I can't stand!
  2. I like to eat mangos.

What is First Order Logic (FOL)?

1. FOL is also called *predicate logic*. A much more expressive language than the propositional logic. It is a powerful language used to develop information about an **object and express the relationship** between objects.
2. FOL not only assumes that does the world contains facts (like PL does), but it also assumes the following:
   1. **Objects**: A, B, people, numbers, colors, wars, theories, squares, pit, etc.
   2. **Relations**: It is unary relation such as red, round, sister of, brother of, etc.
   3. **Function**: father of, best friend, third inning of, end of, etc.

Some of the Examples for First Order Logic Sentences

First Order Logic Sentences

For each of the following English sentences, write a corresponding sentence in FOL.

1. The only good extraterrestrial is a drunk extraterrestrial.
   $\forall x.ET(x) \land Good(x) \rightarrow Drunk(x)$
2. The Barber of Seville shaves all men who do not shave themselves.
   $\forall x.\neg Shaves(x,x) \rightarrow Shaves(BarberOfSeville, x)$
3. There are at least two mountains in England.
   $\exists x,y.Mountain(x) \land Mountain(y) \land InEngland(x) \land InEngland(y) \land x \neq y$
4. There is exactly one coin in the box.
   $\exists x.Coin(x) \land InBox(x) \land \forall y.(Coin(y) \land InBox(y) \rightarrow x = y)$
5. There are exactly two coins in the box.
   $\exists x,y.Coin(x) \land InBox(x) \land Coin(y) \land InBox(y) \land x \neq y \land \forall z.(Coin(z) \land InBox(z) \rightarrow (x = z \lor y = z))$
6. The largest coin in the box is a quarter.
   $\exists x.Coin(x) \land InBox(x) \land Quarter(x) \land \forall y.(Coin(y) \land InBox(y) \land \neg Quarter(y) \rightarrow Smaller(y,x))$
7. No mountain is higher than itself.
   $\forall x.Mountain(x) \rightarrow \neg Higher(x,x)$
8. All students get good grades if they study.
   $\forall x.Student(x) \land Study(x) \rightarrow GetGoodGrade(x)$

Example1:

FOL: Likes(John,IceCream)
English: John likes ice cream.

FOL: Person(Mary)
English: Mary is a person.

FOL: ∀x (Person(x)→Mortal(x))
English: All persons are mortal.

FOL: ∃x (Student(x)∧Enrolled(x,Math))
English: There exists a student who is enrolled in Math.

FOL: FatherOf(John,Mary)
English: John is the father of Mary.

FOL: ∀x ∀y (BrotherOf(x,y)→SiblingOf(x,y))
English: If x is the brother of y, then x is a sibling of y.

FOL: Red(Apple)
English: The apple is red.

FOL: ∃x (Book(x)∧Author(x,J.K. Rowling))
English: There exists a book written by J.K. Rowling.

FOL: Adjacent(Square(1,1),Square(1,2))
English: Square (1,1) is adjacent to Square (1,2).

FOL: ∀x (Animal(x)→∃y ParentOf(y,x))
English: Every animal has a parent.

Example 2:

Here are 25 complex examples of First-Order Logic (FOL) that make use of quantifiers (Universal ∀ and Existential ∃) and logical connectives (such as conjunction ∧ and, disjunction ∨, negation ¬, implication →, and equivalence ↔):

1. ∀x (Human(x)→Mortal(x))

English: All humans are mortal.

2. ∃x (Human(x)∧¬Mortal(x))

English: There exists a human who is not mortal.

3. ∀x (Dog(x)→∃y (OwnerOf(y,x)))

English: Every dog has an owner.

4. ∀x ∃y (Likes(x,y)∧Animal(y))

English: Everyone likes some animal.

5. ∃x (Person(x)∧∀y (Person(y)→Knows(x,y)))

English: There exists a person who knows every other person.

6. ∀x ∃y (BrotherOf(x,y)→¬SiblingOf(x,y))

English: For every person, there exists a brother who is not a sibling of them.

7. ∃x ∀y (Owns(x,y)→Car(y))

English: There exists someone who owns every car.

8. ∀x ∃y (WorksAt(x,y)→City(y))

English: For every person, there exists a city where they work.

9. ∀x (Adult(x)→∃y (ChildOf(y,x)))

English: Every adult has a child.

10. ∃x ∃y (Loves(x,y)∧¬Loves(y,x))

English: There exists someone who loves someone else, but the other person does not love them back.

11. ∀x (Infected(x)→∃y (Doctor(y)∧Treats(y,x)))

English: Every infected person has a doctor who treats them.

12. ∃x ∀y (Owns(x,y)→Car(y))

English: There exists someone who owns every car.

13. ∀x ∃y (ParentOf(y,x)→Human(y))

English: Every child has a parent who is human.

14. ∀x (Student(x)→∃y (Course(y)∧Enrolled(x,y)))

English: Every student is enrolled in at least one course.

15. ∃x (Person(x)∧∀y (Person(y)→Loves(x,y)))

English: There is a person who loves every other person.

16. ∀x (HasCar(x)→∃y (Car(y)∧Owns(x,y)))

English: Every person who has a car owns a car.

17. ∃x ∀y (Animal(x)∧FriendOf(x,y)→Human(y))

English: There exists an animal who is friends with every human.

18. ∀x ∀y (Knows(x,y)→Knows(y,x))

English: If x knows y, then y knows x.

19. ∃x ∃y (City(x)∧City(y)∧¬SameCity(x,y))

English: There exist two cities that are not the same.

20. ∀x (Person(x)→∃y (ParentOf(y,x)∧Human(y)))

English: Every person has a human parent.

21. ∃x ∃y (Loves(x,y)∧¬Loves(y,x))

English: There is someone who loves someone, but that person doesn't love them back.

22. ∀x (Student(x)→∃y (Classroom(y)∧Enrolled(x,y)))

English: Every student is enrolled in a classroom.

23. ∀x ∃y (Animal(x)∧InZoo(x,y))

English: Every animal is in a zoo.

24. ∃x ∃y (BankAccount(x)∧Deposit(y,x))

English: There exists a bank account in which money is deposited.

25. ∀x (Cat(x)→∃y (Pet(y)∧Owns(x,y)))

English : Every cat has a pet that someone owns.

These examples showcase how First-Order Logic uses quantifiers and connectives to form complex and nuanced statements about the world, capturing relationships, properties, and conditions in a formal way.

Basic Elements of FOL

The basic elements of **First-Order Logic (FOL)** are the fundamental components used to express statements and form logical expressions. These elements allow us to describe objects, relations, and functions within a domain of discourse. The key components are as follows:

1. Objects (Domain Elements)

- **Definition**: Objects represent the entities or things in the domain of discourse.
- **Example**: People, animals, numbers, cars, etc.
- **Example in FOL**: John, Alice, 5, Dog.

2. Predicates (Relations or Properties)

- **Definition**: Predicates represent relationships between objects or properties of objects. They are used to express assertions about the domain.
- **Example**: "is a sibling," "is greater than," "is a teacher," "is red."
- **Example in FOL**:
  - Sibling(John,Alice) → "John is a sibling of Alice."
  - GreaterThan(5,3) → "5 is greater than 3."
  - Red(Car) → "The car is red."

3. Constants

- **Definition**: Constants represent specific, fixed objects in the domain of discourse. Unlike variables, constants refer to a particular object and do not change.
- **Example**: John, Alice, 7, Earth, 3.
- **Example in FOL**:
  - Human(John) → "John is a human."
  - Animal(Dog) → "Dog is an animal."

4. Variables

- **Definition**: Variables represent arbitrary objects in the domain of discourse. They are placeholders that can take any value from the domain.
- **Example**: x, y, z.
- **Example in FOL**:
  - Sibling(x,y) → "x is a sibling of y" (where x and y are variables representing any individuals in the domain).

5. Quantifiers

- **Definition**: Quantifiers express how many objects in the domain satisfy a given property. The two main types of quantifiers are:
  - **Universal Quantifier (∀)**: Indicates that a statement is true for all objects in the domain.
  - **Existential Quantifier (∃)**: Indicates that there exists at least one object in the domain for which the statement is true.
- **Examples**:
  - **Universal Quantifier**: ∀x (Human(x)→Mortal(x)) → "All humans are mortal."
  - **Existential Quantifier**: ∃x (Dog(x)∧HasOwner(x))→ "There exists a dog that has an owner."

6. Functions

- **Definition**: Functions map objects to other objects. They are operations or processes that produce a unique result for each input.
- **Example**: "father of," "plus," "best friend," "age of."
- **Example in FOL**:
  - FatherOf(John) → "Father of John."
  - Plus(3,4)=7 → "3 plus 4 equals 7."

7. Logical Connectives

- **Definition**: Logical connectives are used to combine simpler statements into more complex ones. These include:
  - **Negation (¬)**: Represents "not."
  - **Conjunction (∧)**: Represents "and."
  - **Disjunction (∨)**: Represents "or."
  - **Implication (→)**: Represents "if... then..."
  - **Biconditional (↔)**: Represents "if and only if."
- **Examples**:
  - **Negation**: ¬Sibling(John,Alice)→ "John is not a sibling of Alice."
  - **Conjunction**: Sibling(John,Alice)∧Age(John,25) → "John is a sibling of Alice, and John is 25 years old."
  - **Disjunction**: Dog(x)∨Cat(x) → "x is either a dog or a cat."
  - **Implication**: Human(x)→Mortal(x) → "If x is a human, then x is mortal."
  - **Biconditional**: Sibling(x,y)↔Sibling(y,x) → "x is a sibling of y if and only if y is a sibling of x."

8. Terms

- **Definition**: A term is an expression used to refer to objects. Terms can be variables, constants, or function applications.
- **Examples**:
  - **Constant**: John, Alice.
  - **Variable**: x, y.
  - **Function**: FatherOf(John).

9. Atomic Formulas

- **Definition**: An atomic formula is a basic statement in FOL that consists of a predicate applied to terms (which can be objects or variables).
- **Example**:
  - Cat(Tom) → "Tom is a cat."
  - Loves(John,Alice) → "John loves Alice."

10. Well-formed Formula (WFF)

- **Definition**: A well-formed formula is a logical expression that follows the syntax rules of the logic system, using predicates, terms, logical connectives, quantifiers, and variables.
- **Example**:
  - ∀x (Human(x)→Mortal(x)) → "All humans are mortal."

These elements, including **Constants**, allow us to express complex logical statements and build reasoning systems within **First-Order Logic**. Constants are key in identifying specific objects within the domain and help make logical expressions more concrete and specific.

4.1.b Syntax and Semantics of FOL

1. Models for first-order logic
2. Symbols and interpretations
3. Terms
4. Atomic sentences
5. Complex sentences
6. Quantifiers: Universal quantification (∀) /Existential quantification (∃)
7. Equality
8. An alternative semantics? : Data base Semantics

4.1.b.1 Models for FOL

A model in first-order logic is an interpretation that specifies: what each predicate means and the entities that can instantiate the variables.

The entities that can instantiate the variables form the domain of discourse or universe, which is usually required to be a nonempty set.

A model for a first-order language is similar to a truth assignment for propositional logic. It provides all the information needed to determine the truth value of each sentence in the language. Key Characteristics of Models for FOL are as follows:

- They have objects in them!
- The domain of a model is the set of objects or domain elements it contains.
- The domain is required to be nonempty—every possible world must contain at least one object
- The objects in the model may be related in various ways.
- Models in first-order logic require total functions, that is, there must be a value for every input tuple

Example: Consider the figure below which illustrates the model containing five objects, two binary relations, three unary relations and one unary functions.

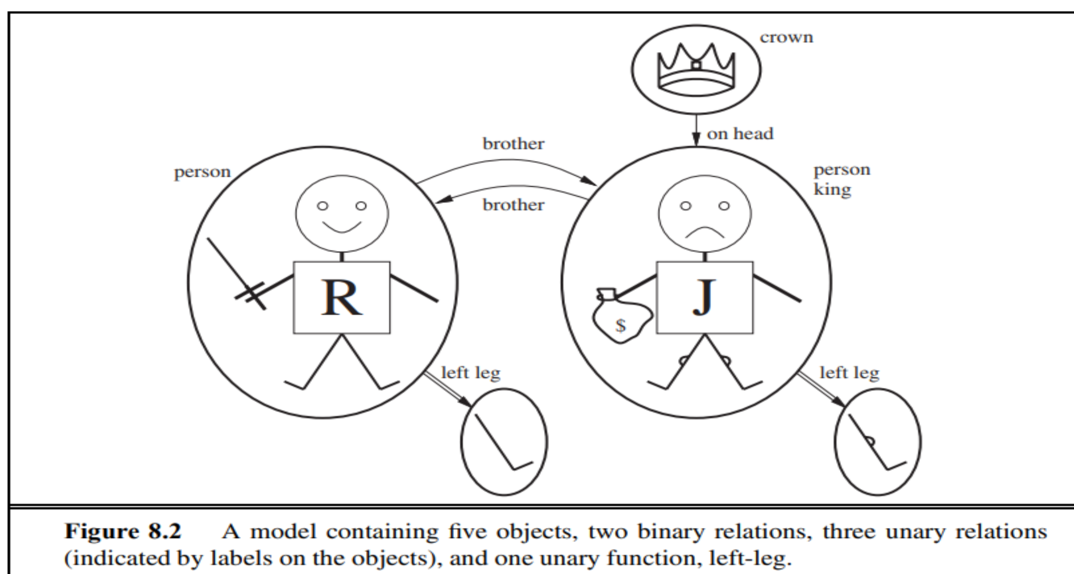

Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

Five objects:

1. Richard the Lionheart, King of England from 1189 to 1199;
2. His younger brother, the evil King John, who ruled from 1199 to 1215;
3. The left legs of Richard and John; and
4. Crown

Tuple: The brotherhood relation in this model is the set { <Richard the Lionheart, King John>, <King John, Richard the Lionheart> } .
Two binary relations: “brother” and “on head” relations are binary relations
Three unary relations/ properties: Person, King and Crown
One unary Function: Left Leg

In summary: A model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, predicate symbols to relations on those objects, and function symbols to functions on those objects. Just as with propositional logic, entailment, validity, and so on are defined in terms of all possible models

4.1.b.2 Symbols and Interpretation

- First-order logic uses symbols to represent: **Objects, Relations and Functions**
- **Types of Symbols**: There are three main types of symbols:

- **Constant Symbols**: Represent specific objects (e.g., **Richard**, **John**).
- **Predicate Symbols**: Represent relationships between objects (e.g., **Brother**, **OnHead**, **Person**, **King**, **Crown**).
- **Function Symbols**: Represent functions that connect objects (e.g., **LeftLeg**).

- **Naming Convention**: All symbols start with uppercase letters.

- **Choice of Names**: Users can choose the names of these symbols freely, similar to how they choose proposition symbols.

- **Arity**: Each predicate and function symbol has an **arity**, which indicates the number of arguments it takes.

- **Models and Interpretations in First-Order Logic:**
  - **Determining Truth**: Just like in propositional logic, every model must provide the information needed to decide if a sentence is true or false.
  - **Components of a Model**: A model consists of:
    - **Objects**: The things being referred to.
    - **Relations**: How these objects are connected.
    - **Functions**: The operations that relate objects to one another.
  - Each model also includes an **interpretation**, which defines what each constant, predicate, and function symbol refers to.

  - **Example of Interpretation**: Here's an example of an interpretation (known as the intended interpretation):
    - **Richard** refers to Richard the Lionheart.
    - **John** refers to the evil King John.
    - **Brother** refers to the relationship of brotherhood.
    - **OnHead** describes the relationship between the crown and King John (i.e., the crown is on King John's head).
    - **Person**, **King**, and **Crown** refer to groups of objects that are people, kings, and crowns, respectively.
    - **LeftLeg** refers to the function that identifies the left leg.

- **Multiple Interpretations**: There are many possible interpretations for the symbols in a model. For example, one interpretation could map **Richard** to the crown and **John** to King John's left leg.

- **Possible Interpretations Count**: If there are **five** objects in a model, there can be **25 possible interpretations** just for the constant symbols **Richard** and **John**.

Not every object needs a name; for instance, the crown and legs might not be named in the intended interpretation.

•**Objects with Multiple Names**: An object can have several names. For example, both **Richard** and **John** could refer to the crown.

•**Understanding Confusion**: This concept can be tricky, but remember that, similar to propositional logic, you can have models where statements like "**Cloudy**" and "**Sunny**" are both true. The **knowledge base** helps eliminate models that don't fit with what we know.

Summary of a Model:

A model in first-order logic includes:

- A set of **objects**.
- An **interpretation** that:
  - Maps **constant** symbols to objects.
  - Maps **predicate** symbols to relations between those objects.
  - Maps **function** symbols to functions involving those objects.

•**Models and Combinations**: Just like in propositional logic, terms like **entailment** and **validity** are defined based on all possible models. Models can have varying numbers of objects, from one to infinity.

- For example, if there are two constant symbols and one object, both symbols must refer to that same object, but this can still apply with more objects.
- If there are more objects than constant symbols, some objects will not have names.

- **Feasibility of Checking Models**:
  - The number of possible models is **unlimited**, making it impractical to check entailment by listing all models in first-order logic.
  - Even when limiting the number of objects, the number of combinations can be very large.
  - For instance, in a case with six or fewer objects, there are **137,506,194,466** possible models.

The syntax of first-order logic with equality, specified in Backus–Naur form

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

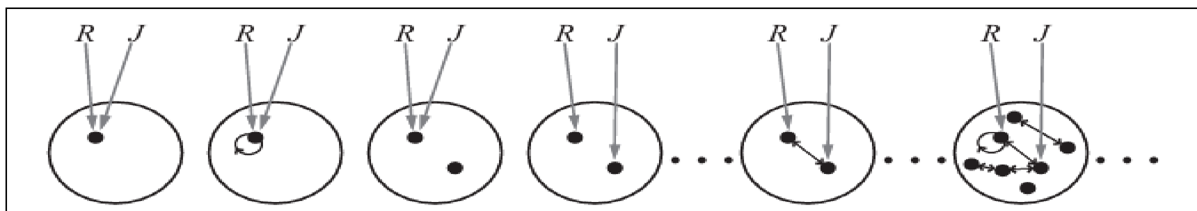

Fig: Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

4.1.b.3: Terms

What is a Term?: A **term** is a logical expression that represents an object. **Constant symbols** are specific types of terms that name particular objects.

Using Function Symbols: Instead of naming every object with a constant symbol, we can use **function symbols**. For example, instead of saying "the left leg of King John," we use **LeftLeg(John)**.

Complex Terms: A **complex term** is created by a function symbol followed by a list of terms in parentheses (e.g., **f(t1, t2)**). Remember, a complex term is just a more complicated name; it is **not** a function or a subroutine that gives back a value.

Reasoning About Terms: We can discuss concepts like "everyone has a left leg" without needing to define what **LeftLeg** means.This reasoning is different from programming, where you need a defined subroutine to return a value.

Understanding the Semantics of Terms: The semantics of a term like **f(t1, t2,...,tn)** works as follows:

- The function symbol **f** refers to a specific function in the model (let's call it **F**).
- The argument terms (like **t1**, **t2**) refer to objects in the domain (let's call them **d1**, **d2**).
- The entire term refers to the object that results from applying the function **F** to the objects **d1**, **d2**, etc.

• **Example**: If **LeftLeg** refers to a function in our model, and **John** refers to King John, then **LeftLeg(John)** represents King John's left leg.

• **Role of Interpretation**: The **interpretation** helps determine what each term refers to within the model.

4.1.b.4: Atomic Sentences

Combining Terms and Predicate Symbols: We can combine **terms** (for objects) and **predicate symbols** (for relations) to create **atomic sentences**.

What is an Atomic Sentence?: An **atomic sentence** (or **atom**) is made up of:

- A predicate symbol
- An optional list of terms in parentheses

Example: Brother(Richard, John) states that Richard the Lionheart is the brother of King John.

Complex Terms in Atomic Sentences: Atomic sentences can also use **complex terms** as arguments.

Example: **Married(Father(Richard), Mother(John))** means that Richard's father is married to John's mother.

Truth of Atomic Sentences: An atomic sentence is **true** in a model if: The relation described by the predicate symbol holds true for the objects referred to by the terms.

4.1.b.5: Complex Sentences

Building Complex Sentences: We can use **logical connectives** (like NOT, AND, OR, and IMPLIES) to create more complex sentences in first-order logic. These connectives follow the same rules (syntax and semantics) as in **propositional calculus**.

Examples of Complex Sentences: Here are four sentences that are **true** in a specific model based on our intended interpretation:

1. **¬Brother(LeftLeg(Richard), John)**: It is **not true** that Richard's left leg is the brother of John.
2. **Brother(Richard, John) ∧ Brother(John, Richard)**: Richard is the brother of John **and** John is the brother of Richard.
3. **King(Richard) ∨ King(John)**: Either Richard **is** a king **or** John **is** a king.
4. **¬King(Richard) ⇒ King(John)**: If Richard **is not** a king, then John **is** a king.

4.1.b.6: Quantifiers: Universal quantification (∀) /Existential quantification (∃)

Understanding Quantifiers: Quantifiers help express properties of groups of objects instead of naming each one. There are **two main types of quantifiers** in first-order logic:

- **Universal Quantifier (∀)**: Indicates a property applies to all objects.
- **Existential Quantifier (∃)**: Indicates a property applies to at least one object.

Universal Quantification (∀): It allows us to state general rules. For example: The statement "All kings are persons" is written as:

- **∀x King(x) ⇒ Person(x)**

This means: "For all x, if x is a king, then x is a person."
The symbol **x** is a **variable** that represents any object, and it is usually a lowercase letter.

Existential Quantification (∃): This quantifier is used to state that something exists without naming it. For example: The statement "King John has a crown on his head" can be expressed as:

  - **∃x Crown(x) ∧ OnHead(x, John)**
  - This means: "There exists an x such that x is a crown and x is on King John's head."

The symbol **∃x** is pronounced "There exists an x such that..."

Nested Quantifiers: We can use multiple quantifiers together, either of the same type or mixed. For example:

  - "Brothers are siblings" can be written as:**∀x ∀y Brother(x,y) ⇒ Sibling(x,y)**
  - "Everybody loves somebody" means: **∀x ∃y Loves(x,y)** (for every person x, there exists someone y that x loves).
  - "There is someone who is loved by everyone" means:**∃y ∀x Loves(x,y)** (there exists someone y who is loved by every person x).

Importance of Order: The order of quantifiers matters, For example:

- **∀x (∃y Loves(x,y))**: Every person loves someone.
- **∃y (∀x Loves(x,y))**: There is someone who is loved by everyone.

Using the same variable name for different quantifiers can be confusing. It's best to use different names.

Connections Between ∀ and ∃: Universal and existential quantifiers are linked through negation:

- "Everyone dislikes parsnips" is equivalent to "There is no one who likes parsnips":
  - **∀x ¬Likes(x, Parsnips) ≡ ¬∃x Likes(x, Parsnips)**.
- "Everyone likes ice cream" is equivalent to "There is no one who does not like ice cream":
  - **∀x Likes(x, IceCream) ≡ ¬∃x ¬Likes(x, IceCream)**.

These relationships follow **De Morgan's Laws** for quantified sentences:

- **∀x ¬P ≡ ¬∃x P**
- **¬∀x P ≡ ∃x ¬P**
- **∀x P ≡ ¬∃x ¬P**
- **∃x P ≡ ¬∀x ¬P**
- **¬(P ∨ Q) ≡ ¬P ∧ ¬Q**
- **¬(P ∧ Q) ≡ ¬P ∨ ¬Q**

Quantifiers are essential for expressing general rules and properties about objects in first-order logic. Understanding how to use universal (∀) and existential (∃) quantifiers, their nesting, and their connection **through negation is crucial for effective reasoning in logic.**

4.1.b.7: Equality

Using Equality in Atomic Sentences: In first-order logic, we can create atomic sentences using the **equality symbol** (=) to indicate that two terms refer to the same object.

- Example: **Father(John) = Henry** means that the object represented by Father(John) is the same as the object represented by Henry.

Determining Truth of Equality: The truth of an equality sentence is determined by checking if the two terms refer to the same object in a given interpretation.

Negation of Equality: The equality symbol can also be used with negation (¬) to assert that two terms are **not** the same object.

- Example: To express that Richard has at least two brothers, we can write: **∃x, y Brother(x, Richard) ∧ Brother(y, Richard) ∧ ¬(x = y)**

This states that there exist two different brothers (x and y) of Richard.

Common Mistake: The sentence **∃x, y Brother(x, Richard) ∧ Brother(y, Richard)** does not correctly indicate that Richard has at least two brothers. This sentence could be true even if Richard only has one brother.

- For example, if both x and y are assigned to King John, the sentence would still be true.

Importance of Negating Equality: The addition of **¬(x = y)** ensures that x and y are different, which is necessary to confirm that Richard has at least two brothers.

Abbreviation: Sometimes, the notation $\mathbf{x \neq y}$ is used as shorthand for saying $\neg(\mathbf{x = y})$. However, it's important to be clear about their meanings.

Equality in first-order logic is a powerful tool for stating facts about objects. By using the equality symbol, we can express when two terms refer to the same object or ensure that they refer to different ones. Being careful with the use of negation is essential for accurately conveying relationships between objects.

4.1.b.8: An alternative semantics? : Data base Semantics

Expressing Relationships: Suppose we want to express that Richard has two brothers, John and Geoffrey. We might write:

Brother (John, Richard) ∧ Brother (Geoffrey, Richard).
However, this statement has some problems:

- It could be true even if Richard only has one brother.
- It doesn't exclude the possibility of Richard having more brothers besides John and Geoffrey.

Correct Expression: To accurately convey that "Richard's brothers are John and Geoffrey," we need a more detailed expression:

Brother(John, Richard) ∧ Brother(Geoffrey, Richard) ∧ John ≠ Geoffrey ∧ ∀x Brother(x, Richard) ⇒ (x = John ∨ x = Geoffrey).

This ensures: John and Geoffrey are indeed the only brothers of Richard.

Challenges with First-Order Logic: This detailed expression is longer and more complex than how we naturally speak, making it easy to make mistakes when translating knowledge into first-order logic. Such errors can lead to unexpected results in logical reasoning systems.

Database Semantics Proposal: To simplify logical expressions, a method called **database semantics** is proposed, which involves three key ideas:

- **Unique-Names Assumption**: Each constant symbol refers to a different object.
- **Closed-World Assumption**: If something is not known to be true, it is assumed to be false.
- **Domain Closure**: A model includes only the objects mentioned by the constant symbols—no additional objects.

Benefits of Database Semantics:

- Under this system, the earlier expression correctly indicates that Richard's brothers are John and Geoffrey.

- Database semantics is also commonly used in logic programming and database systems.

Possible Models: In database semantics, there are limited possible models for a situation. For instance:

- With two objects, there are 16 different combinations of relationships that can satisfy the conditions, much fewer than the infinite possibilities in standard first-order logic.

Choosing the Right Approach: There is no single "correct" way to interpret logic. The best choice depends on:

- o How clear and simple it is to express the knowledge.
- o How easy it is to create logical rules from that knowledge.

Database semantics works well when we are sure of the identities of all objects and have all relevant facts. However, it can be tricky when details are unclear.

Figure below shows some of the models, ranging from the model with no tuples satisfying the relation to the model with all tuples satisfying the relation. With two objects, there are four possible two-element tuples, so there are 24 =16 different subsets of tuples that can satisfy the relation. Thus, there are 16 possible models in all—a lot fewer than the infinitely many models for the standard first-order semantics.

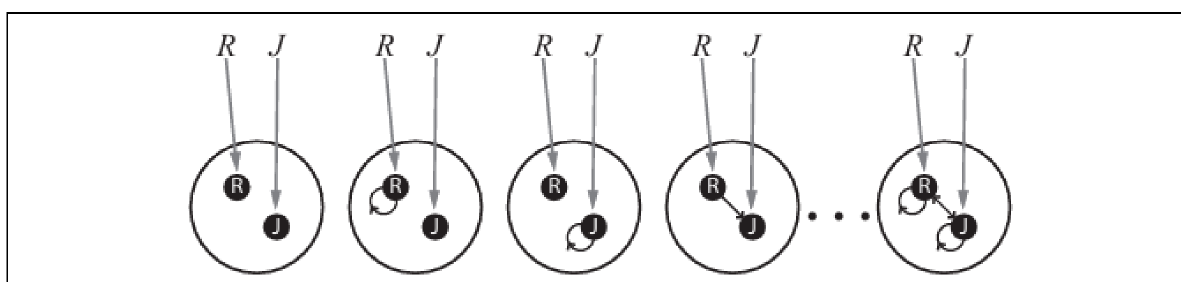

Fig: Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol, under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

4.1.c Using First Order Logic

Purpose of First-Order Logic: Now that we have a detailed logical language, we can start using it effectively. The best way to understand this is by looking at examples.

Understanding Domains: In knowledge representation, a **domain** is simply a specific area or topic about which we want to express knowledge.

Examples of domains: family relationships, numbers, sets, lists, and the Wumpus world.

TELL/ASK Interface: Before exploring domains, we will introduce the **TELL/ASK interface** for interacting with a knowledge base:

- **TELL**: Adds new information to the knowledge base.
- **ASK**: Queries the knowledge base to retrieve information.

Example Domains: We'll examine different simple domains in turn:

- Family relationships
- Numbers
- Sets and lists
- The Wumpus world (a common problem-solving environment)

Advanced Example: In the next section, we'll look at a more detailed example: **electronic circuits**.

4.1.c.1 Assertions and Queries in First-Order Logic

Assertions (TELL): Statements we add to a knowledge base (KB) using **TELL**. These statements are called **assertions**.

- Example assertions:
  - **TELL(KB, King(John))** – asserts that John is a king.
  - **TELL(KB, Person(Richard))** – asserts that Richard is a person.
  - **TELL(KB, ∀x King(x) ⇒ Person(x))** – asserts that all kings are people.

Queries (ASK):

- **ASK** lets us check if something is true in the KB. These questions are called **queries** or **goals**.
  - Example: **ASK(KB, King(John))** returns true if John is indeed a king in the KB.

- **Logical Entailment**: Queries that follow logically from the KB should return true.
  - Example: If we've asserted that John is a king and that all kings are people, then **ASK(KB, Person(John))** will also return true.

Quantified Queries:

We can use **ASK** with existential quantifiers to see if there exists an example that fits the query.

- Example: **ASK(KB, ∃x Person(x))** returns true if there is at least one person in the KB.

However, this is a general answer and doesn't provide the specific values that make the query true.

ASKVARS: **ASKVARS** helps us find specific values (substitutions) that satisfy a query.

- Example: **ASKVARS(KB, Person(x))** will return {x/John} and {x/Richard}, showing both John and Richard satisfy the query.
- **Substitution (Binding List)**: This list of values for variables that makes the query true is called a **substitution** or **binding list**.

Limitations in First-Order Logic:

- ASKVARS works well with KBs made only of Horn clauses, where variables can always bind to specific values to make queries true.
- In first-order logic, some queries may be true without binding any variables.

  Example: If **TELL(KB, King(John) ∨ King(Richard))** is in KB, then **ASK(KB, ∃x King(x))** is true, but we don't have a specific binding for x.

4.1.c.2 The Kinship Domain in First-Order Logic

Kinship Domain Basics:

- This domain represents family relationships like parenthood, siblinghood, marriage, etc.
- Objects are **people**, and we use predicates to represent relationships and attributes.

Predicates:

- **Unary predicates** (describe one property about a person):
  - **Male(x)**, **Female(x)**
- **Binary predicates** (describe relationships between two people):
  - **Parent (x, y)**, **Sibling (x, y)**, **Brother (x, y)**, **Sister (x, y)**, **Child (x, y)**, **Daughter (x, y)**, **Son (x, y)**, **Spouse (x, y)**, **Wife (x, y)**, **Husband (x, y)**, **Grandparent (x, y)**, **Grandchild (x, y)**, **Cousin (x, y)**, **Aunt (x, y)**, **Uncle(x, y)**

Functions for Unique Relationships:

- Functions like **Mother(x)** and **Father(x)** define unique relationships (e.g., every person has exactly one mother and one father).

Defining Relationships:

- **Mother**: "One's mother is one's female parent."
  - Example: **∀m, c Mother(c) = m ⇔ Female(m) ∧ Parent(m, c)**
- **Husband**: "One's husband is one's male spouse."
  - Example: **∀w, h Husband(h, w) ⇔ Male(h) ∧ Spouse(h, w)**
- **Mutually Exclusive**: "Male and Female are disjoint (a person can't be both)."
  - Example: **∀x Male(x) ⇔ ¬Female(x)**
- **Parent and Child Relationship**:
  - Example: **∀p, c Parent(p, c) ⇔ Child(c, p)**
- **Grandparent**: "A grandparent is a parent of one's parent."
  - Example: **∀g, c Grandparent (g, c) ⇔ ∃p Parent (g, p) ∧ Parent (p, c)**
- **Sibling**: "A sibling is another child of the same parents."
  - Example: **∀x, y Sibling (x, y) ⇔ x ≠ y ∧ ∃p Parent (p, x) ∧ Parent (p, y)**

Axioms and Theorems:

- **Axioms** are basic, foundational statements about relationships. For example, the rule defining siblings is an axiom.
- **Theorems** are statements that can be logically derived from axioms.
  - Example: Sibling relationships are symmetric (if A is a sibling of B, then B is a sibling of A): **∀x, y Sibling(x, y) ⇔ Sibling(y, x)**

Including Facts and Missing Axioms:

- Not all axioms define concepts; some provide **general information** without being complete definitions (e.g., **Person(x)** doesn't have a full definition).
- **Facts** like **Male(Jim)** or **Spouse(Jim, Laura)** can help solve specific questions (like confirming family relationships).
- If expected answers aren't obtained, it often means an **axiom is missing** from the system.

4.1.c.3 Numbers, Sets, and Lists in First-Order Logic

Natural Numbers and Peano Axioms

Natural Numbers: These are non-negative whole numbers (like 0, 1, 2, 3, etc.) that we can define using simple rules, called the *Peano axioms*, which help explain how numbers and addition work.
To define natural numbers, we need:

- A rule, **NatNum**, that tells us if something is a natural number.
- The number **0** as a starting point.
- A rule called **S (successor)**, which helps us create the next number by adding 1 each time.

The **Peano Axioms** define natural numbers as follows:

1. **0 is a natural number**: We say **NatNum(0)** is true.
2. **Each natural number has a successor**: If **n** is a natural number, then **S(n)** (which is **n + 1**) is also a natural number.

So, starting from 0, we get **0, S(0), S(S(0)),...** which gives **0, 1, 2,...**.
Additional rules (or axioms) for successors:

- **0 is not the successor of any number**: This helps us prevent backward counting (e.g., **0 ≠ S(n)**).
- **If two numbers are equal, their successors are also equal**: If **m = n**, then **S(m) = S(n)**.

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:

- $\forall m\ NatNum(m) \Rightarrow +(0,m)=m$.
- $\forall m,n\ NatNum(m) \wedge NatNum(n) \Rightarrow +(S(m),n) = S(+(m,n))$.

Defining Addition with the Successor Function

To define **addition** with the Peano rules:

1. **Adding 0 to any natural number keeps it the same**: For any number **m**, **m + 0 = m**.
2. **Adding a successor**: If we add **S(m)** (or **m + 1**) to **n**, it is the same as taking **S(m + n)**.

We can also use **infix notation** for readability, which means writing **m + n** instead of +(**m, n**).
For example:

- **m + 1** means we are adding 1 to m.
- The addition axiom using infix notation looks like **(m + 1) + n = (m + n) + 1**.

This builds addition as a repeated application of the successor function.

Infix Notation and Syntactic Sugar

- **Infix Notation as Syntactic Sugar**: Using symbols like + in expressions (e.g., **m** + **n**) is an example of "syntactic sugar." Syntactic sugar is a way to simplify or abbreviate the language we use without changing the actual meaning (or semantics). For instance, we could write +(**m, n**) instead of **m** + **n**, but **m** + **n** is easier to read.
- **Desugaring**: Any statement using syntactic sugar can be rewritten (or "desugared") back into the more basic, formal logic form without losing its meaning.

Building Up Math Concepts

Once we define **addition** with Peano axioms, we can easily build other math operations:

- **Multiplication** as repeated addition.
- **Exponentiation** as repeated multiplication.

- We can also define **division**, **remainders**, and even concepts like **prime numbers**.

So, all of **number theory** can be constructed using just one **constant** (0), one **function** (successor), one **predicate** (NatNum), and **four axioms**. This foundation supports advanced applications like **cryptography**.

Sets and Set Theory

- **Sets**: Sets are also fundamental in math and common sense. Using sets, we can define **number theory** and represent collections of elements, including the **empty set**. We build up sets by adding elements, finding intersections (common elements), and unions (combined elements).
- **Set Theory Notation as Syntactic Sugar**: We use familiar symbols in set theory as syntactic sugar for clarity. For instance:
  - **{}** represents the empty set.
  - **Set(x)** tells us if something is a set.
  - **x ∈ s** means **x** is an element of set **s**.
  - **s1 ⊆ s2** means set **s1** is a subset of **s2**.
  - **s1 ∩ s2** and **s1 ∪ s2** are the intersection and union of sets **s1** and **s2**, respectively.
  - **{x | s}** represents a new set created by adding **x** to **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:

$$\forall s \;\; Set(s) \;\Leftrightarrow\; (s = \{\,\}) \lor (\exists x, s_2 \;\; Set(s_2) \land s = \{x|s_2\}) \,.$$

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 s_2 adjoined with some element y, where either y is the same as x or x is a member of s_2:

$$\forall x, s \;\; x \in s \;\Leftrightarrow\; \exists y, s_2 \;\; (s = \{y|s_2\} \land (x = y \lor x \in s_2)) \,.$$

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:

$$\forall s_1, s_2 \;\; s_1 \subseteq s_2 \;\Leftrightarrow\; (\forall x \;\; x \in s_1 \;\Rightarrow\; x \in s_2) \,.$$

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

$$\forall s_1, s_2 \;\; (s_1 = s_2) \;\Leftrightarrow\; (s_1 \subseteq s_2 \land s_2 \subseteq s_1) \,.$$

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

$$\forall x, s_1, s_2 \;\; x \in (s_1 \cap s_2) \;\Leftrightarrow\; (x \in s_1 \land x \in s_2) \,.$$

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

$$\forall x, s_1, s_2 \;\; x \in (s_1 \cup s_2) \;\Leftrightarrow\; (x \in s_1 \lor x \in s_2) \,.$$

Lists

- **Definition of Lists**:
  - Lists are similar to sets but have some key differences:
    - **Ordered**: The order of elements in a list matters.
    - **Duplicates**: The same element can appear multiple times in a list.
- **Vocabulary Used**:
  - **Nil**: Represents an empty list (no elements).
  - **Cons**: A function used to add an element to the front of a list.
  - **Append**: A function used to join two lists together.
  - **First**: A function that retrieves the first element of a list.
  - **Rest**: A function that retrieves all elements of a list except the first one.
  - **Find**: A predicate that checks if an element is in a list (similar to how Member works for sets).
  - **List?**: A predicate that checks if something is a list.
- **Syntactic Sugar**:
  - Just like with sets, we often use simpler notation when writing about lists:
    - The empty list is written as `[]`.
    - `Cons(x, y)` (where `y` is a non-empty list) is written as `[x|y]`.
    - `Cons(x, Nil)` (the list containing only the element `x`) is written as `[x]`.
    - A list with several elements, such as `[A, B, C]`, corresponds to the nested structure `Cons(A, Cons(B, Cons(C, Nil)))`.

4.1.c.4 Wumpus World

Overview of the Wumpus World:

- The Wumpus World is a common example used in artificial intelligence to illustrate how an agent can perceive its environment and make decisions based on that information.
- Propositional logic axioms were discussed in Chapter 7, but first-order axioms provide a more concise and clear way to express the necessary information.

Agent Perception:

- The Wumpus agent receives a **percept vector** with five elements, indicating what it perceives in the environment.
- Each percept must be recorded with the time it was observed to avoid confusion. For example:
  - **Example Percept Sentence**: `Percept([Stench, Breeze, Glitter, None, None], 5)`
    - This means at time 5, the agent perceives a stench, breeze, and glitter, with no other percepts.
- **Actions** in the Wumpus World can be represented logically:
  - **Actions**: `Turn(Right), Turn(Left), Forward, Shoot, Grab, Climb`

Determining Best Action:

- The agent uses a query to find the best action to take:
  - **Example Query**: `ASKVARS(∃a BestAction(a, 5))`
  - This returns a binding list, like `{a/Grab}`, indicating the agent should grab.

Logical Implications:

- The percept data leads to certain facts about the current state:
  - **Example Sentences**:
    - `∀t,s,g,m,c Percept([s, Breeze, g, m, c], t) ⇒ Breeze(t)`
    - `∀t,s,b,m,c Percept([s, b, Glitter, m, c], t) ⇒ Glitter(t)`
- This is a simple reasoning process called **perception** and will be explored further in Chapter 24.

Reflex Behavior:

- Reflex actions can be implemented using quantified implications:
  - **Example**: ∀t Glitter(t) ⇒ BestAction(Grab, t)
    - This implies if the agent perceives glitter at time t, the best action is to grab.

Representing the Environment:

- **Objects** in the Wumpus World include squares, pits, and the Wumpus.
- Instead of naming each square (e.g., Square1, Square2), we use lists to represent their coordinates:
  - **Example of Adjacency**:
    - ∀x,y,a,b Adjacent([x,y],[a,b]) ⇔ (x = a ∧ (y = b-1 ∨ y = b+1)) ∨ (y = b ∧ (x = a-1 ∨ x = a+1))
- **Pits**: Use a unary predicate Pit to indicate where pits are located.
- **Wumpus**: Represented simply by a constant, Wumpus.
- The agent's location is tracked over time:
  - **Example**: At(Agent, s, t) means the agent is at square s at time t.
- The Wumpus's location can be fixed:
  - **Example**: ∀t At(Wumpus, [2,2], t)

Properties of Squares:

- An object can only occupy one location at a time:
  - **Example**: ∀x,s1,s2,t At(x,s1,t) ∧ At(x,s2,t) ⇒ s1 = s2
- From its location and percepts, the agent can infer properties of its environment:
  - **Example**:
    - If the agent is at a square and perceives a breeze, then that square is breezy:
    - ∀s,t At(Agent, s, t) ∧ Breeze(t) ⇒ Breezy(s)

Deducing Locations:

- By identifying breezy and non-breezy squares, the agent can infer where pits and the Wumpus are located:

   - **Example Axiom**: `∀s Breezy(s) ⇔ ∃r Adjacent(r,s) ∧ Pit(r)`
- First-order logic allows for more concise rules without needing separate axioms for each square or time step:
   - **Example**:
      - For the arrow: `∀t HaveArrow(t+1) ⇔ (HaveArrow(t) ∧ ¬Action(Shoot, t))`

The first-order logic framework allows for more efficient and clear representations of the Wumpus World, making it easier to manage the agent's perception, actions, and the environment.

4.1.d Knowledge Engineering in First Order Logic

The previous section demonstrated how to use first-order logic to represent knowledge in three simple areas. This section focuses on the overall process of constructing a knowledge base, known as **knowledge engineering**. A knowledge engineer is a person who studies a specific domain, identifies key concepts, and creates a formal representation of the objects and relationships within that domain.

To illustrate the knowledge engineering process, we will use an electronic circuit domain that should already be somewhat familiar, allowing us to focus on the representation aspects. The method we use is suitable for creating specialized knowledge bases with clearly defined domains and known types of queries.

Knowledge engineering projects can vary in many ways, but they all generally include the following steps:

1. **Identify the Task**: The knowledge engineer defines what questions the knowledge base will answer and what facts are needed for each problem. For example, in the wumpus world, does the knowledge base need to decide on actions (like moving or grabbing) or just provide information about the environment (like the presence of pits or wumpus)? This step is similar to the PEAS process for designing agents discussed in Chapter 2.
2. **Assemble Relevant Knowledge**: The knowledge engineer may already be an expert in the field or may need to collaborate with experts to gather the necessary information, a process known as knowledge acquisition. At this stage, knowledge is not formally represented. The goal is to understand the knowledge base's scope based on the task and how the domain functions.

  - For the wumpus world, which has a defined set of rules, identifying relevant knowledge is straightforward. For example, the adjacency of squares needs to be known, but this was not explicitly stated in the wumpus-world rules. In real-world domains, determining what knowledge is relevant can be complex, such as deciding if a VLSI simulation needs to consider stray capacitances and skin effects.
3. **Decide on Vocabulary**: The engineer translates important concepts into logical names, including predicates, functions, and constants. This involves stylistic decisions that can significantly affect the project's success. For example, should pits be represented as objects or as a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? The choices made form the **ontology** of the domain, which describes what exists without detailing their specific properties or relationships.
4. **Encode General Knowledge**: The knowledge engineer writes axioms for all vocabulary terms. This clarifies the meanings and allows experts to verify the content. This step often uncovers misunderstandings or gaps, requiring a return to step 3 for revisions.

   For example, if the knowledge base includes a diagnostic rule for finding the wumpus, such as:

   ∀s Smelly(s)⇒Adjacent(Home(Wumpus),s)

   If this is not a biconditional, the agent will never be able to prove the absence of wumpuses.

5. **Encode Specific Problem Instances**:
   - With a well-thought-out ontology, this step should be straightforward. It involves creating simple atomic sentences about instances of concepts already in the ontology. For example, if the wumpus is located at a specific square, the engineer would represent this as a statement like **At(Wumpus, [2, 2], t)** for some time t. For logical agents, problem instances come from sensors, while a "disembodied" knowledge base receives additional sentences like traditional programs receive input data.
6. **Pose Queries and Get Answers**:
   - This is the rewarding part, where the inference procedure uses the axioms and specific facts to derive useful information. For example, the engineer might ask, "What is the best action to take at time 5?" This step eliminates the need for a custom solution algorithm, as the knowledge base can derive conclusions on its own.
7. **Debug the Knowledge Base**:

- Initially, the answers to queries may not match expectations. While the answers reflect the knowledge base's content, they may not be what the user anticipates. For example, if an axiom is missing, some queries may not be answerable. Debugging may involve identifying gaps or weak axioms by recognizing where reasoning stops unexpectedly. Missing or weak axioms can lead to incomplete conclusions; for example, a statement like: **∀xNumOfLegs(x,4)⇒Mammal(x)** is false for reptiles, amphibians, and, more importantly, tables. The falsehood of this sentence can be determined independently of the rest of the knowledge base. In contrast, a typical programming error, like: **offset=position+1**, cannot be assessed without context, such as whether "offset" refers to the current position or the next one, or if the value of "position" changes elsewhere in the program.

To better understand this seven-step process, we will apply it to an extended example in the domain of electronic circuits.

8.4.2 The Electronic Circuits Domain

In this section, we will create an ontology and knowledge base to help us understand digital circuits, following the seven steps of knowledge engineering.

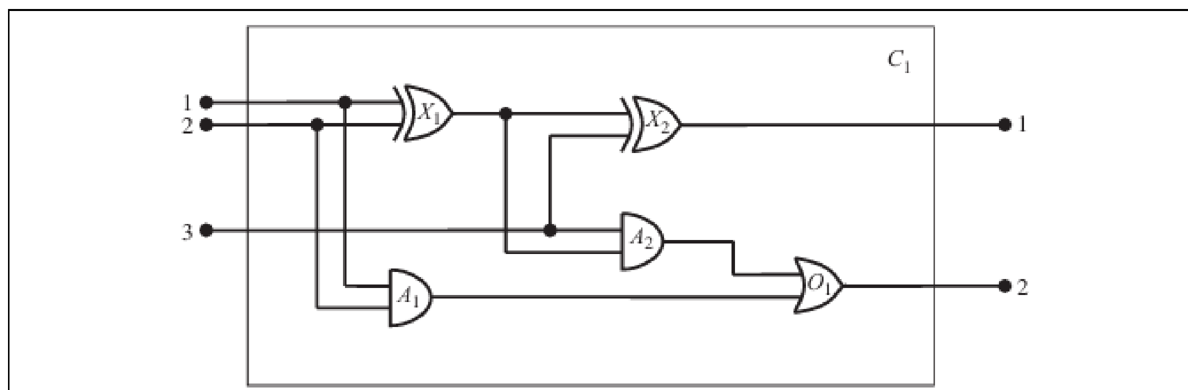

Fig : A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two ANDgates, and one OR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

1. Identify the Task

We need to analyze digital circuits, like the one shown in Figure 8.6. Our main tasks include:

- Checking if the circuit adds correctly.
- Finding out what the output of gate A2 is when all inputs are high.
- Identifying which gates are connected to the first input terminal.
- Looking for feedback loops in the circuit. There are also more detailed analyses involving timing delays, circuit area, power consumption, and production cost. Each of these requires additional knowledge.

2. Assemble Relevant Knowledge

Digital circuits consist of wires and gates.

- Signals travel along wires to the input terminals of gates, which then produce output signals on another terminal.
- Gates can be of four types: AND, OR, XOR (each with two inputs), and NOT (with one input).
- We focus on the connections between terminals, not the wires themselves. For our analysis, the size, shape, and cost of components are not relevant.
- If we were debugging faulty circuits, we would need to consider wires, since a faulty wire can affect the signals.

3. Decide on Vocabulary

We will discuss circuits, terminals, signals, and gates. Here's how we will represent them:

- **Gates**: Each gate is represented as an object with a name, e.g., `Gate(X1)`, and its type is defined using a function, like `Type(X1)=XOR`.
- **Circuits**: Represented by a predicate, e.g., `Circuit(C1)`.
- **Terminals**: Identified using a predicate, e.g., `Terminal(x)`. Each gate can have input and output terminals, denoted by functions `In(1,X1)` and `Out(1,X1)`.
- **Connectivity**: Represented by a predicate `Connected`, which connects terminals, e.g., `Connected(Out(1,X1),In(1,X2))`.
- **Signal Values**: We can use a predicate `On(t)` to check if a signal is on, but it's easier to use objects `1` and `0` with a function `Signal(t)` to represent signal values.

4. Encode General Knowledge of the Domain

We can establish clear rules for our ontology. Here are the key axioms:

1. **If two terminals are connected, they have the same signal:**
   $$\forall t_1, t_2 \;\; Terminal(t_1) \land Terminal(t_2) \land Connected(t_1, t_2) \Rightarrow Signal(t_1) = Signal(t_2) .$$
2. **Each terminal's signal is either 1 or 0**
   $$\forall t \;\; Terminal(t) \Rightarrow Signal(t) = 1 \lor Signal(t) = 0$$
3. **Connectivity 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 outputs 0 if any input 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 outputs 1 if any input 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 outputs 1 if 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. **AND, OR, and XOR gates have two inputs and one output; NOT gates have one input 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 capacity:**

$$\forall c,i,j \;\; Circuit(c) \land Arity(c,i,j) \;\Rightarrow\; \forall n \;\; (n \leq i \;\Rightarrow\; Terminal(In(c,n))) \land (n > i \;\Rightarrow\; In(c,n) = Nothing) \land \forall n \;\; (n \leq j \;\Rightarrow\; Terminal(Out(c,n))) \land (n > j \;\Rightarrow\; Out(c,n) = Nothing)$$

11. **Gates, terminals, signals, and gate types 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. **All gates are also circuits:**

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

5.Encode the Specific Problem Instance

We describe the circuit from Figure as circuit C1:

- **Circuit:** **`Circuit(C1) ∧ Arity(C1,3,2)`**
- **Gates:**
  - **`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`**

- **Connections:**

$$
\begin{array}{ll}
Connected(Out(1,X_1), In(1,X_2)) & Connected(In(1,C_1), In(1,X_1)) \\
Connected(Out(1,X_1), In(2,A_2)) & Connected(In(1,C_1), In(1,A_1)) \\
Connected(Out(1,A_2), In(1,O_1)) & Connected(In(2,C_1), In(2,X_1)) \\
Connected(Out(1,A_1), In(2,O_1)) & Connected(In(2,C_1), In(2,A_1)) \\
Connected(Out(1,X_2), Out(1,C_1)) & Connected(In(3,C_1), In(2,X_2)) \\
Connected(Out(1,O_1), Out(2,C_1)) & Connected(In(3,C_1), In(1,A_2)) .
\end{array}
$$

6. Pose Queries to the Inference Procedure

What combinations of inputs would cause the first output of C_1 (the sum bit) to be 0 and the second output of C_1 (the carry bit) to be 1?

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

The answers are substitutions for the variables i_1, i_2, and i_3 such that the resulting sentence is entailed by the knowledge base. ASKVARS will give us three such substitutions:

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

What are the possible sets of values of all the terminals for the adder circuit?

$$
\exists i_1, i_2, i_3, o_1, o_2 \;\; Signal(In(1,C_1)) = i_1 \wedge Signal(In(2,C_1)) = i_2 \\
\wedge\, Signal(In(3,C_1)) = i_3 \wedge Signal(Out(1,C_1)) = o_1 \wedge Signal(Out(2,C_1)) = o_2 .
$$

This final query will return a complete input–output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of circuit verification.

7. Debug the Knowledge Base

We can perturb the knowledge base in various ways to see what kinds of erroneous behaviors emerge. For example, suppose we fail to read Section 8.2.8 and hence forget to assert that 1≠ 0. Suddenly, the system will be unable to prove any outputs for the circuit, except for the input cases 000 and 110. We can pinpoint the problem by asking for the outputs of each gate. For example, we can ask

$$\exists\, i_1, i_2, o \;\; Signal(In(1, C_1)) = i_1 \wedge Signal(In(2, C_1)) = i_2 \wedge Signal(Out(1, X_1)) \; ,$$

which reveals that no outputs are known at X_1 for the input cases 10 and 01. Then, we look at the axiom for XOR gates, as applied to X_1:

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; Signal(In(1, X_1)) \neq Signal(In(2, X_1)) \; .$$

If the inputs are known to be, say, 1 and 0, then this reduces to

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; 1 \neq 0 \; .$$

Now the problem is apparent: the system is unable to infer that $Signal(Out(1, X_1)) = 1$, so we need to tell it that $1 \neq 0$.

4.2 Inference in First Order Logic

4.2.1 Propositional Versus First Order Inference

a) Inference rules for quantifiers
b) Reduction to propositional inference

4.2.1.a) Inference Rules for Quantifiers

Let us begin with universal quantifiers. Suppose our knowledge base contains the standard folkloric axiom stating that 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)) .**
- ---------------------

a) The rule of Universal Instantiation (UI for short)

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.

To write out the inference rule formally, we use SUBST(θ, α) denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v \;\; \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

for any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

b) The rule for Existential Instantiation

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,

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

For example, from the sentence

$$\exists x \ \ Crown(x) \land OnHead(x, John)$$

we can infer the sentence

$$Crown(C_1) \land OnHead(C_1, John)$$

as long as C_1 does not appear elsewhere in the knowledge base.

Existential Sentences and Existential Instantiation:

Existential Sentences: These sentences state that there is at least one object that meets a certain condition.

Existential Instantiation: This rule lets us assign a unique name to an object that meets the condition in an existential sentence. However, this name should not already belong to any other object.

Example from Mathematics: Suppose we discover a number slightly larger than 2.71828 that satisfies the equation $\frac{d(x^y)}{dy} = x^y$ for a specific value of x. We might name this number "e." However, if we were to use an existing name, such as "π," for it, this would cause confusion because "π" is already assigned to a different mathematical constant.

Skolem Constants: In logic, the new name we give to satisfy an existential condition is called a *Skolem constant*. This is part of a larger process called *skolemization*.

Application Rules:

- **Universal Instantiation**: Can be applied repeatedly to derive various conclusions.
- **Existential Instantiation**: Applied only once. Afterward, we can discard the original existentially quantified sentence.

Example of Discarding an Existential Sentence: Suppose we have the existential sentence **∃x Kill(x, Victim),** which states that someone killed the victim. By Existential Instantiation, we could assign the name "**Murderer**" to the unknown person, resulting in the sentence **Kill(Murderer, Victim).** At this point, we no longer need **∃x Kill(x, Victim)** because we now have a specific term, **"Murderer,"** to refer to the person who committed the act.

Inferential Equivalence: Although the knowledge base isn't exactly the same after applying Existential Instantiation, it remains ***inferentially equivalent***. This means the revised knowledge base will be satisfiable under the same conditions as the original one.

4.2.1.b) Reduction to propositional inference (Propositionalization)

Once we have rules for inferring non quantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. By applying rules that derive non quantified (propositional) sentences from quantified ones, we can simplify first-order inference to propositional inference, making it easier to solve. The first idea is that **Replacing Quantified Sentences as follows**:

- **Existential Quantifiers**: A single instantiation replaces an existentially quantified sentence.
- **Universal Quantifiers**: A universally quantified sentence is replaced by all possible instantiations.

Example: Given a knowledge base with:

  - **∀x (King(x) ∧ Greedy(x) ⇒ Evil(x))**
  - **King(John)**
  - **Greedy(John)**
  - **Brother(Richard, John)**

- By applying Universal Instantiation (UI) to the quantified sentence, we substitute each variable x with ground terms (like "John" or "Richard"):
  - **King(John) ∧ Greedy(John) ⇒ Evil(John)**
  - **King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)**
- After this, we can treat each atomic sentence (e.g., King(John), Greedy(John)) as a proposition and apply propositional logic techniques to make inferences, such as concluding Evil(John).

Complete Propositionalization: Every first-order knowledge base can be propositionalized so that entailment is preserved. This method makes it possible to determine if a sentence can be entailed (i.e., proven) from a first-order knowledge base.

Infinite Instantiations Problem: If the knowledge base includes function symbols, the ground-term substitutions could be infinite (e.g., using Father could produce terms like Father(Father(Father(John)))). This makes it difficult for propositional algorithms to handle an infinitely large set of sentences.

Herbrand's Theorem (1930): Jacques Herbrand proved that if a sentence is entailed by the knowledge base, only a finite subset of these substitutions is needed for the proof. We can organize the process by generating terms gradually:

- Start with constant symbols (e.g., Richard, John).
- Move to terms of depth 1 (e.g., Father(Richard)).
- Then terms of depth 2, and so on, until a proof is found.

Completeness of the Approach: This propositionalization approach is *complete*, meaning any entailed sentence can be proved even though the space of possible models is infinite.

Challenges – Semidecidability:

- If a sentence is not entailed, there is no way to know for sure; the proof process might continue indefinitely.
- This resembles the *halting problem* in Turing machines, where we can't always know if a process will end.
- The entailment problem in first-order logic is **semidecidable**:
  - Algorithms exist that confirm when a sentence is entailed (saying "yes").
  - However, no algorithm can reliably say "no" for every non-entailed sentence.

Example : For example, suppose our knowledge base contains just the sentences

FOL Inference

$\forall x \;\; King(x) \land Greedy(x) \;\Rightarrow\; Evil(x)$
$King(John)$
$Greedy(John)$
$Brother(Richard, John)$.

Propositional logic Inference

$King(John) \land Greedy(John) \;\Rightarrow\; Evil(John)$
$King(Richard) \land Greedy(Richard) \;\Rightarrow\; Evil(Richard)$,

Modus Ponens

Modus Ponens, also known as **"the law of detachment"**, is a fundamental rule of logic in propositional reasoning. It states that:

If a conditional statement is true (e.g., "If P, then Q") and the antecedent (P) is true, then the consequent (Q) must also be true.

Formal Representation

1. **Premise 1**: $P \rightarrow Q$ (If P then Q)
2. **Premise 2**: P (P is true)
3. **Conclusion**: Q (Therefore, Q is true)

Example

1. **Premise 1**: If it rains, the ground will be wet. ($P \rightarrow Q$)
2. **Premise 2**: It rains. (P)
3. **Conclusion**: The ground will be wet. (Q)

Why It's Important

Modus Ponens is essential because it allows us to draw conclusions from known facts and rules. It is widely used in:

- Mathematics
- Computer Science
- Philosophy
- Artificial Intelligence

4.2.2 Unification (and Lifting)

- Generalized Modus Ponens is a lifted version of Modus Ponens—it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic.
- **Generalized Modus Ponens**: For atomic sentences pi , pi ′ , and q, where there is a substitution θ

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

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

There are $n+1$ premises to this rule: the n atomic sentences $p_i{}'$ and the one implication. The conclusion is the result of applying the substitution θ to the consequent q. For our example:

$p_1{}'$ is $King(John)$ $\quad$ p_1 is $King(x)$
$p_2{}'$ is $Greedy(y)$ $\quad$ p_2 is $Greedy(x)$
θ is $\{x/John, y/John\}$ $\quad$ q is $Evil(x)$
$\text{SUBST}(\theta, q)$ is $Evil(John)$.

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a **unifier** for them if one exists:

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

Example : Suppose we have a query **AskVars(Knows(John, x))**: whom does John know? Answers to this query can be found by finding all sentences in the knowledge base that unify with **Knows(John, x)**. Here are the results of unification with four different sentences that might be in the knowledge base:

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)}
UNIFY(Knows(John, x), Knows(x,Elizabeth)) = fail .

In first-order logic**, unification is** a process used to find a common instantiation for two predicates or terms such that they become identical.

- It's a fundamental operation in logic programming and automated reasoning, allowing for the comparison and integration of different logical expressions.
- Unification is essential for tasks such as theorem proving, pattern matching, and resolution in logic-based systems.

A substitution, on the other hand, is a mapping of variables to terms.

- It's essentially a set of assignments that replaces variables in logical expressions with specific terms, thereby creating a new expression that may be simpler or more specific than the original one.
- Substitutions are used to represent the results of unification and are crucial for maintaining consistency and correctness in logical inference.

Unification Algorithm

```
function UNIFY(x, y, θ) 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
          θ, the substitution built up so far (optional, defaults to empty)

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

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

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).

Unification is the process of finding a substitution that makes two logical expressions identical. The algorithm takes two expressions, x and y, and attempts to find a substitution (θ) that makes them identical.

Here's a breakdown of how the algorithm works:

Base case: If the substitution **θ i**s already marked as a failure, then it returns failure immediately.

Identity check: If **x** and **y** are identical, it means no further unification is needed, and the current substitution **θ** can be returned.

Variable check: If **x** is a variable, it calls the **UNIFY-VAR** function with **x** as the variable and **y** as the expression. If **y** is a variable, it calls **UNIFY-VAR** with y as the variable and x as the expression.

Compound expression check: If both **x** and **y** are compound expressions, it recursively calls **UNIFY** on their arguments and operators.

List check: If both **x** and **y** are lists, it recursively calls UNIFY on their first elements and their remaining elements.

Failure case: If none of the above conditions are met, it returns failure, indicating that x and y cannot be unified.

The UNIFY-VAR function is used when one of the expressions **(x or y)** is a variable. It attempts to create a substitution based on the variable and the expression it's being unified with.

Substitution check: If the substitution already contains a mapping for the variable, it recursively calls UNIFY with the mapped value and the expression x.

Reverse substitution check: If the expression is already in the substitution, it recursively calls UNIFY with the variable and the mapped value.

Occur check: Checks for a possible occurrence of the variable in the expression, preventing infinite loops, and returns failure if such an occurrence is detected.

Substitution addition: If none of the above cases apply, it adds a new mapping to the substitution, indicating that the variable is unified with the expression.

Overall, the algorithm systematically traverses through the

- expressions,
- handling variables,
- Compound statements,
- lists, and
- checking for failures,

- until it either finds a *successful substitution* or determines that unification is *not possible*.

Example: Suppose we have the following two predicates:

1. **Predicate P(x,y)**
2. **Predicate Q(f(z),a)**

Here,

- P and Q are predicates,
- x, y, and z are variables, and
- f and a are constants.

Now, let's say we want to **unify P(x,y) with Q(f(z),a).**

We can use the given algorithm for unification to find a substitution that makes these two predicates identical.

1. **Initially, θ is empty**.
2. **Start unifying the predicates**: P(x,y) and Q(f(z),a)
   Since P and Q are different, they can't be unified directly.
3. **Unify the arguments**: Unify x with f(z) and y with a
4. **Unify x with f(z)**:
   - x is a variable, f(z) is a compound term.
   - Call UNIFY-VAR(x, f(z), θ):
     - Add x/f(z) to θ
     - θ={x/f(z)}
5. **Unify y with a**:
   - y is a variable, a is a constant.
   - Call UNIFY-VAR(y, a, θ):
     - Add y/a to θ
     - θ={x/f(z),y/a}
6. **Finally, return θ**:
   θ={x/f(z),y/a}

So, the resulting substitution θ makes P(x,y) and Q(f(z),a) identical:

P(x,y){x/f(z),y/a}=Q(f(z),a)

Rules for unification in First-Order Logic (FOL):

---

1. If both terms are identical, unification succeeds.

- Example: a=a or P(x,y)=P(x,y)
  - No substitution is required, as the terms are already identical.

---

2. If one term is a variable, unify it with the other term.

- **Variable Rule:** A variable can be unified with:
  - A constant (e.g., x=a)
  - Another variable (e.g., x=y)
  - A function or compound term (e.g., x=f(y))
- **Conditions:**
  - The variable must not appear in the term it is unified with (to prevent circular definitions, called the **occurs-check**).
- **Example:**
  - x and f(a) unify as {x→f(a)}.
  - x and x unify with no substitution needed.

---

3. If both terms are constants, unification succeeds only if they are the same.

- Example:
  - a and a: Succeeds.
  - a and b: Fails (different constants).

---

4. If both terms are compound expressions (functions or predicates), unify their components.

- **Conditions:**
  - The outer operators (function or predicate names) must match.
  - The number of arguments must be the same.
  - Their arguments must unify recursively.
- **Example:**
  - f(x,b) and f(a,y):

        - Unify x with a ({x→a}.
        - Unify b with y ({y→b}.
        - Result: {x→a,y→b}.

---

5. If both terms are lists, unify their elements pair by pair.

- Unify the first elements, then unify the rest of the lists recursively.
- **Example:**
    - [x,y] and [a,b]:
        - Unify x with a.
        - Unify y with b.
        - Result: {x→a,y→b}.

---

6. Unification fails if:

- Two terms are of different types:
    - Example: A constant and a function (e.g., a and f(x).
- Their structures are incompatible:
    - Example: f(a,b) and g(a,b) (different operators).
- A variable would need to unify with itself through a compound term (occurs-check):
    - Example: x and f(x) fail to unify (circular dependency).

Summary of Unification Rules:

1. **Identical Terms:** No substitution needed.
2. **Variable Unification:** Substitute the variable with the other term, ensuring no circular dependency.
3. **Constant Unification:** Succeed only if they are the same.
4. **Compound Unification:** Match operators and unify arguments recursively.
5. **List Unification:** Unify elements pair by pair.
6. **Failure Cases:** Occurs-check violations, incompatible types, or mismatched structures.

Storage and retrieval:

Underlying the **TELL** and **ASK** functions used to inform and interrogate a knowledge base are the more primitive **STORE** and FETCH functions. STORE(s) stores a sentence s into the knowledge base and FETCH(q) returns all unifiers such that the query q unifies with some.

Given a sentence to be stored, it is possible to construct indices for all possible queries that unify with it. For the fact Employs(IBM , Richard), the queries are

Employs(IBM , Richard) Does IBM employ Richard?
Employs(x, Richard) Who employs Richard?
Employs(IBM , y) Whom does IBM employ?
Employs(x, y) Who employs whom?

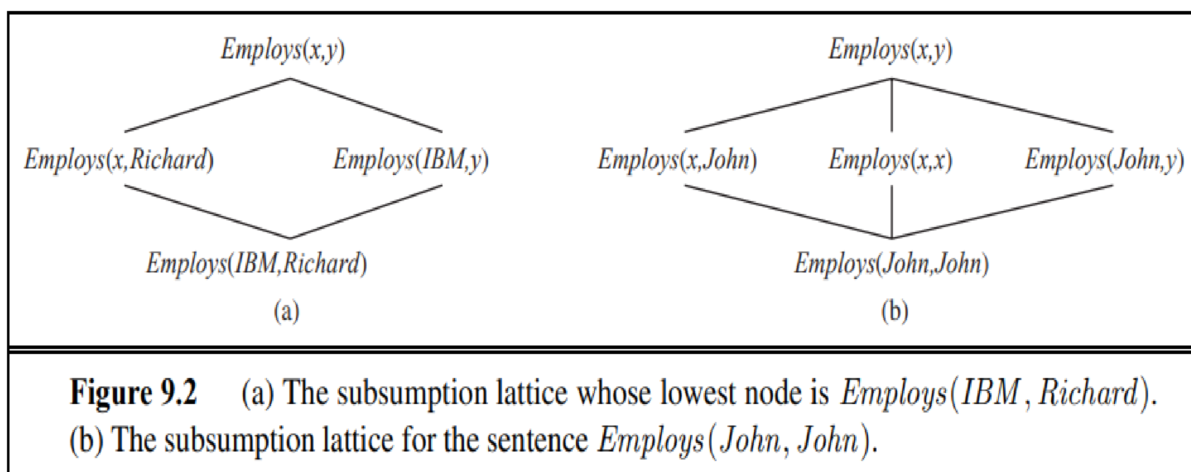

Figure 9.2 (a) The subsumption lattice whose lowest node is *Employs*(*IBM* , *Richard*). (b) The subsumption lattice for the sentence *Employs*(*John*, *John*).

These queries form a subsumption lattice, as shown in Figure 9.2(a). The lattice has some interesting properties. For example, the child of any node in the lattice is obtained from its parent by a single substitution; and the "highest" common descendant of any two nodes is the result of applying their most general unifier. The portion of the lattice above any ground fact can be constructed systematically (Exercise 9.5). A sentence with repeated constants has a slightly different lattice, as shown in Figure 9.2(b). Function symbols and variables in the sentences to be stored introduce still more interesting lattice structures.

4.2.3 Forward Chaining

Forward chaining is a reasoning method , starts with the known facts and uses inference rules to derive new conclusions until the goal is reached or no further inferences can be made.
In essence, it proceeds forward from the premises to the conclusion.

Example : Consider the following knowledge base representing a simple diagnostic system:

1. ***If a patient has a fever, it might be a cold.***
2. ***If a patient has a sore throat, it might be strep throat.***
3. ***If a patient has a fever and a sore throat, they should see a doctor.***

Given the facts:

- **The patient has a fever.**
- **The patient has a sore throat.**

- **Forward chaining would proceed as follows:**

1. **Check the first rule: Fever? Yes. Proceed.**
2. **Check the second rule: Sore throat? Yes. Proceed.**
3. **Apply the third rule: The patient has a fever and sore throat, thus they should see a doctor.**

Forward chaining is suitable for situations where there is a large amount of known information and the goal is to derive conclusions.
Forward chaining 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.
First-order definite clauses : 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. The following are first-order definite clauses:

- King(x) ∧ Greedy(x) ⇒ Evil(x) .
- King(John) .
- Greedy(y) .

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified.
Consider the following problem: *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 Colonel West, who is American.

We will prove that West is a criminal.

First, we will represent these facts as first-order definite clauses.

1. **"... 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) .**
2. **"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)**
     - **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. **and we must know that 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) .**

From these inferred facts, we can conclude that Colonel West is indeed a criminal since he sold missiles to a hostile nation, which is Nono.
"... it is a crime for an American to sell weapons to hostile nations":
American(West) ∧ Weapon(Missile) ∧ Sells(West, Missile, Nono) ∧ Hostile(Nono) ⇒ Criminal(West) .

DATALOG :

- 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.

- Datalog gets its name because it can represent the type of statements typically made in relational databases.

A simple forward-chaining algorithm

function FOL-FC-ASK(KB, α) **returns** a substitution or *false*
  inputs: KB, the knowledge base, a set of first-order definite clauses
    α, 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 θ 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(θ, q)
        if q' does not unify with some sentence already in KB or new **then**
          add q' to new
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not $fail$ **then return** ϕ
    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.

Explanation of Algorithm :

This algorithm is an implementation of Forward Chaining with a goal-directed query mechanism, specifically designed for First-Order Logic (FOL) knowledge bases.

It's called Forward Chaining with Ask (FOL-FC-ASK). Let's break down the steps:

Algorithm:

1. **Inputs:**
    - **KB**: The knowledge base, which consists of a set of first-order definite clauses.
    - **α:** The query, which is an atomic sentence.
2. **Loop until no new sentences are inferred:**
    - Initialize **new** as an empty set.
3. **Iterate through each rule in the knowledge base**:

- Standardize the variables in the rule (**ensuring variable names are unique**).
- For each **substitution θ** that makes the antecedent of the rule (`p1 ∧ ... ∧ pn`) match some subset of the KB:
  1. Apply the substitution to the consequent of the rule (`q`) to generate a new sentence **`q'`**.
  2. Check if **`q'`** unifies with some sentence already in the KB or `new`. If not, add `q'` to `new`.
  3. Attempt to unify **`q'`** with the query **`α`**. If unification succeeds (resulting in a substitution ϕ), return ϕ.
  4. **Update the knowledge base**:
     - Add the sentences in `new` to the KB
  5. **Repeat the loop** until no new sentences are inferred or until the query is proven or disproven.

4. **Output**:

- If the query is proven, return the substitution that makes it true.
- If the query is disproven (i.e., it cannot be proven true), return false.

Detailed Explanation of Forward Chaining Algorithm with Example

The provided algorithm is a **Forward Chaining** implementation for first-order logic (FOL). It attempts to derive a query (α) from a given knowledge base (KB) of definite clauses using substitutions.

Explanation of the Algorithm

1. **Inputs:**
   - KB: A set of first-order definite clauses (facts and rules).
   - α: The query, an atomic sentence.
2. **Initialization:**
   - new: A set of new inferences made during each iteration.
3. **Outer Loop:** Repeats until no new sentences are inferred:
   - Initialize new as an empty set.
4. **Inner Loop:** For each rule in KB:
   - The rule is of the form $(p_1 \land ... \land p_n) \Rightarrow q$.
   - Standardize the variables in the rule to avoid conflicts.
5. **Inference Check:** For every possible substitution θ:
   - Check if θ makes $(p_1 \land ... \land p_n)$ true using facts from the KB.
   - If true:
     - Apply θ to the conclusion q to get q'.
     - If q' is not already in KB or new, add it to new.

6. **Unification Check:** For the current inference `q'`:
   - Check if `q'` unifies with the query α.
   - If unification is successful, return the substitution ϕ.
7. **Update Knowledge Base:**
   - Add all new inferences (`new`) to `KB`.
8. **Termination:**
   - If no inference unifies with the query α, return `false`.

Example1

Knowledge Base (KB):

1. `Parent(John, Mary)`
2. `Parent(Mary, Alice)`
3. `Parent(x, y) ∧ Parent(y, z) ⇒ Grandparent(x, z)`

Query (α):

`Grandparent(John, Alice)`

Step-by-Step Execution:

1. **Iteration 1:**
   - **Rule:** `Parent(x, y) ∧ Parent(y, z) ⇒ Grandparent(x, z)`
   - **Possible substitutions**:
     - For `x=John`, `y=Mary` from `Parent(John, Mary)`.
     - For `y=Mary`, `z=Alice` from `Parent(Mary, Alice)`.
     - Combined substitution `θ = {x=John, y=Mary, z=Alice}` satisfies the premise.
   - **Conclusion**: Apply `θ` to `Grandparent(x, z)` → `Grandparent(John, Alice)`.
   - Add `Grandparent(John, Alice)` to `new`.
2. **Unification:**
   - `Grandparent(John, Alice)` unifies with α.
   - Return substitution `ϕ = {x=John, z=Alice}`.

Output:

The algorithm returns the substitution `{x=John, z=Alice}`, indicating that John is indeed the grandparent of Alice.

Key Points:

- Forward chaining starts with facts and rules in the `KB`, making inferences step-by-step until it proves or disproves the query.
- It is **data-driven**, unlike backward chaining, which is goal-driven.

Example2

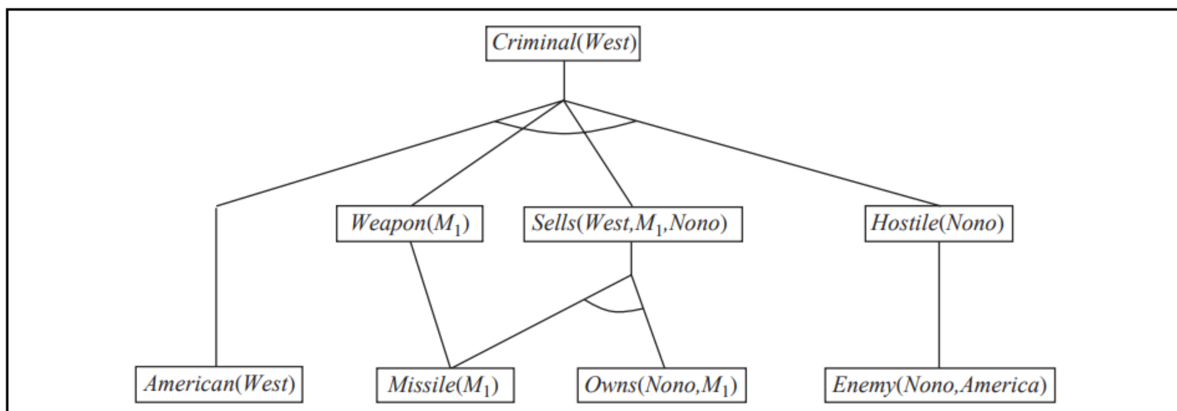

Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

Example3:

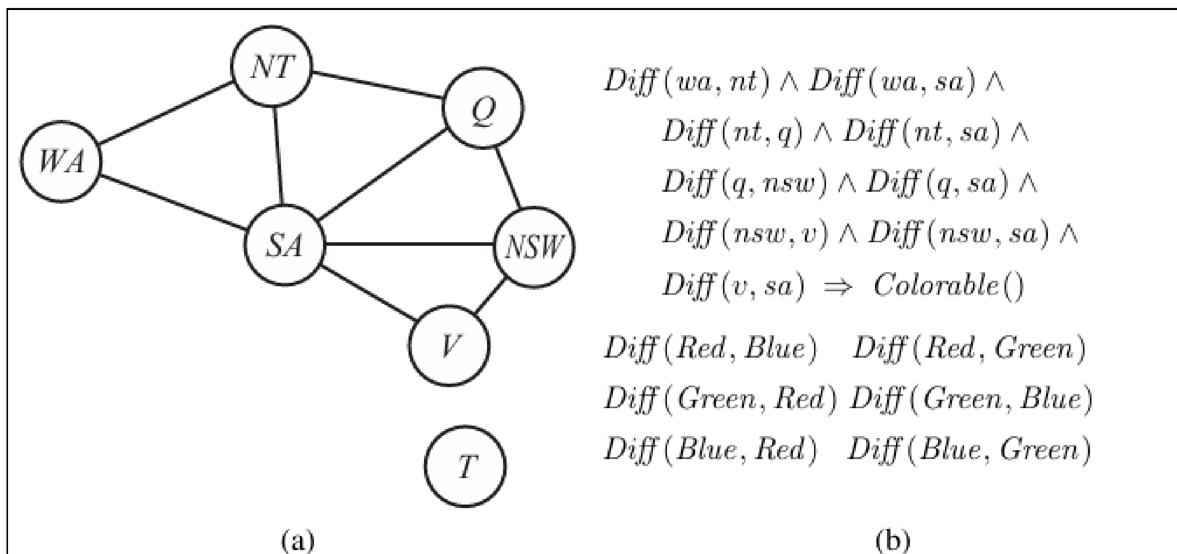

Figure 9.5 (a) Constraint graph for coloring the map of Australia. (b) The map-coloring CSP expressed as a single definite clause. Each map region is represented as a variable whose value can be one of the constants *Red*, *Green* or *Blue*.

The connection between pattern matching and constraint satisfaction is indeed very strong. Each conjunct can be interpreted as a constraint on the variables it encompasses; for instance, Missile(x) acts as a unary constraint on x. Building on this concept, we can represent every finite-domain constraint satisfaction problem (CSP) as a single definite clause alongside some associated ground facts.

Consider the map-coloring problem illustrated in Figure 9.5(a). An equivalent formulation as a single definite clause is provided in Figure 9.5(b). It is clear that the conclusion Colorable() can only be inferred if the CSP has a solution. Since CSPs generally include 3-SAT problems as specific cases, we can deduce that matching a definite clause against a set of facts is NP-hard.

It may appear discouraging that forward chaining involves an NP-hard matching problem in its inner loop. However, there are three ways to alleviate this concern:

1. We can remind ourselves that most rules in real-world knowledge bases are small and simple, similar to the rules in our crime example, rather than large and complex like the CSP formulation in Figure 9.5. In database contexts, it is common to assume that both the sizes of rules and the arities of predicates are limited by a constant. Thus, the focus shifts to data complexity—the complexity of inference based on the number of ground facts in the knowledge base. It is relatively easy to show that the data complexity of forward chaining is polynomial.
2. We can examine subclasses of rules where matching is efficient. Essentially, every Datalog clause can be regarded as defining a CSP, meaning that matching will be tractable when the corresponding CSP is tractable. For example, if the constraint graph—the graph with variables as nodes and constraints as edges—forms a tree, the CSP can be solved in linear time. This same result applies to rule matching. For instance, if we remove South Australia from the map in Figure 9.5, the resulting clause can be expressed as:

   Diff(wa,nt)∧Diff(nt,q)∧Diff(q,nsw)∧Diff(nsw,v)⇒Colorable()

This corresponds to the reduced CSP depicted in Figure 6.12 on page 224. Algorithms designed for solving tree-structured CSPs can be directly applied to the problem of rule matching.

3. We can work on eliminating redundant rule-matching attempts in the forward-chaining algorithm, as will be detailed next.

Incremental Forward Chaining Summary

Incremental forward chaining improves upon traditional forward chaining by avoiding redundant rule matching. In the original forward chaining example, rules could repeatedly match against known facts, such as the rule **Missile(x) ⇒ Weapon(x)** matching against Missile(M1) even when Weapon(M1) was already inferred. To eliminate this redundancy, an incremental algorithm checks a rule

only if its premise includes a conjunct that unifies with a newly inferred fact from the previous iteration. This method maintains efficiency while generating the same set of facts.

With appropriate indexing, rules that can be triggered by a given fact can be easily identified, allowing for a system to update dynamically as new facts are introduced. Typically, only a small subset of rules is activated by a new fact, which means a lot of unnecessary work occurs when constructing partial matches with unsatisfied premises. For instance, a partial match is created on the first iteration but discarded when it fails to succeed until the next iteration. It is more efficient to retain and complete these partial matches as new facts come in.

The Rete algorithm addresses this issue by pre-processing rules into a dataflow network where each node represents a literal from a rule premise. This network captures variable bindings, allowing for the filtering of matches and reducing recomputation. Rete networks have become crucial in production systems—early forward-chaining systems like XCON (R1)—which utilized thousands of rules to configure computer components for customers.

Production systems are also implemented in cognitive architectures, where they model human reasoning. In these systems, productions interact with working memory, allowing for real-time operation even with millions of rules.

A final inefficiency in forward chaining arises from its intrinsic nature, as it generates all allowable inferences based on known facts, regardless of their relevance to the current goal. For example, if a knowledge base contains unrelated facts about eating habits and missile prices, irrelevant conclusions may arise. To mitigate this, alternatives such as backward chaining can be used, or forward chaining can be restricted to a subset of rules. Additionally, in deductive databases, forward chaining is tailored to only consider relevant variable bindings through a technique called magic sets. This approach rewrites the rule set using goal-related information to constrain the variables being considered, focusing the inference process on pertinent facts and enhancing efficiency.

XXXXXXXXXXXXXXXXXX_End of the Module No: 4_XXXXXXXXXXXXXXXXXXXX