# Module 4

First Order Logic and Inferences in FOL

# Topics

1. **Unification:** *is a process used to find a common instantiation for two predicates or terms such that they become identical.*
2. **Forward Chaining:** *is a reasoning and inference procedure which starts with known facts and moves forward to reach conclusions*
3. **Backward Chaining:** *is a reasoning and inference procedure which starts with the goal and moves backward to verify if the goal can be satisfied,*
4. **Resolution:** *is an inference rule used to derive new clauses by combining existing ones***.**

These ***techniques are essential for reasoning and inference in First-Order Logic systems***.

# 1. Unification

- In first-order logic**, unification is** a process used to **find a common instantiation for two predicates or terms such that they become identical**.
- **A substitution**, on the other hand, is a **mapping of variables** to terms.

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

**Examples :**

**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 .**

# 1. Unification

- In first-order logic**, unification is** a process used to **find a common instantiation for two predicates or terms such that they become identical**.
- **A substitution**, on the other hand, is a **mapping of variables** to terms.

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

**Examples :**

**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 .**

**UNIFY(Knows(John, x), Knows(x17,Elizabeth)) = {x/Elizabeth, x17/John}**

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

  **if** $\theta$ = failure **then return** failure
  **else if** $x = y$ **then return** $\theta$
  **else if** VARIABLE?($x$) **then return** UNIFY-VAR($x, y, \theta$)
  **else if** VARIABLE?($y$) **then return** UNIFY-VAR($y, x, \theta$)
  **else if** COMPOUND?($x$) **and** COMPOUND?($y$) **then**
    **return** UNIFY($x$.ARGS, $y$.ARGS, UNIFY($x$.OP, $y$.OP, $\theta$))
  **else if** LIST?($x$) **and** LIST?($y$) **then**
    **return** UNIFY($x$.REST, $y$.REST, UNIFY($x$.FIRST, $y$.FIRST, $\theta$))
  **else return** failure

---

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

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

**Figure 9.1** The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution $\theta$ 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.

**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)**

# 2. 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: Another Example

- **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) .**

# Forward Chaining Algorithm [ Reading Exercise]

```
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 ← { }
      for each rule in KB do
          (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
          for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                        for some p'1, ..., p'n in KB
              q' ← SUBST(θ, q)
              if q' does not unify with some sentence already in KB or new then
                  add q' to new
                  φ ← 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.

# 3.Backward Chaining

- Backward chaining is a reasoning method that starts with **the goal and works backward through the inference rules** to find out whether the goal can be satisfied by the known facts.
- It's essentially **goal-driven reasoning,** where the system seeks to prove the hypothesis by breaking it down into subgoals and verifying if the premises support them.

**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.**

- **Backward chaining would proceed as follows:**
  1. **Start with the goal**: Should the patient see a doctor?
  2. **Check the third rule**: Does the patient have a cold and a sore throat? **Yes.**
  3. **Check the first and second rules**: Does the patient have a fever and sore throat? **Yes**.
  4. **The goal is satisfied**: The patient should see a doctor.
- Backward chaining is useful when there is a specific goal to be achieved, and the system can efficiently backtrack through the inference rules to determine whether the goal can be satisfied.

# Backward Chaining Algorithm [ Reading Exercise]

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ′ in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ′

generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    first,rest ← FIRST(goals), REST(goals)
    for each θ′ in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ″ in FOL-BC-AND(KB, rest, θ′) do
        yield θ″
```

**Figure 9.6** A simple backward-chaining algorithm for first-order knowledge bases.

# 4. Resolution

- Resolution is based on the principle of proof by contradiction.
- Resolution combines logical sentences in the form of clauses to derive new sentences.
- The resolution rule states that if there are two clauses that contain complementary literals (**one positive, one negative**) then these literals can be resolved, leading to a new clause that is inferred from the original clauses.

# Example:

**Consider two logical statements:**

1. PVQ
2. ¬PVR

**Applying resolution:** Resolve the statements by eliminating P:

- PVQ
- ¬PVR
- Resolving P and ¬P: **QVR**

The resulting statement **QVR is a new clause** inferred from the original two. Resolution is a key component of **logical reasoning in FOL**, especially in tasks like automated theorem proving and knowledge representation.

# 4. Resolution

- Conjunctive normal form for first-order logic : As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form (CNF)—**that is, a conjunction of clauses, where each clause is a disjunction of literals.
- Literals can contain variables, which are assumed to be universally quantified. For example, the sentence
- **∀ x American(x) ∧ Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z) ⇒ Criminal(x)** becomes, in CNF,
- **¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x, y, z) ∨ ¬Hostile(z) ∨ Criminal(x)** .

# Resolution

- Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence.
- The procedure for conversion to CNF is similar to the propositional case, The principal difference arises from the need to eliminate existential quantifiers.
- **We illustrate the procedure by translating the sentence**
- “Everyone who loves all animals is loved by someone,” or
- **$\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x, y)] \Rightarrow [\exists y \text{ Loves}(y, x)]$** .

# Steps

- **Eliminate implications**: ∀ x [¬∀ y ¬Animal(y) ∨ Loves(x, y)] ∨ [∃ y Loves(y, x)] .
- **Move ¬ inwards**: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have
  - ¬∀ x p becomes ∃ x ¬p
  - ¬∃ x p becomes ∀ x ¬p .
- **Our sentence goes through the following transformations**:
  - ∀ x [∃ y ¬(¬Animal(y) ∨ Loves(x, y))] ∨ [∃ y Loves(y, x)] .
  - ∀ x [∃ y ¬¬Animal(y) ∧ ¬Loves(x, y)] ∨ [∃ y Loves(y, x)] .
  - ∀ x [∃ y Animal(y) ∧ ¬Loves(x, y)] ∨ [∃ y Loves(y, x)] .
- **Standardize variables**: For sentences like (∃ x P(x))∨(∃ x Q(x)) which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have
  - ∀ x [∃ y Animal(y) ∧ ¬Loves(x, y)] ∨ [∃ z Loves(z, x)] .

- **Skolemize**: Skolemization is the process of removing existential quantifiers by elimination. Translate ∃ x P(x) into P(A), where A is a new constant.
  - **Example :**
    - ∀ x [Animal(A) ∧ ¬Loves(x, A)] ∨ Loves(B, x) ,
    - ∀ x [Animal(F(x)) ∧ ¬Loves(x, F(x))] ∨ Loves(G(z), x) . Here F and G are Skolem functions.
- **Drop universal quantifiers**: At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers:
  - [Animal(F(x)) ∧ ¬Loves(x, F(x))] ∨ Loves(G(z), x) .
- **Distribute ∨ over ∧:**

[Animal(F(x)) ∨ Loves(G(z), x)] ∧ [¬Loves(x, F(x)) ∨ Loves(G(z), x)] .

# The resolution inference rule

- Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain **complementary literals**. Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one ***unifies with the negation of the other***.
- Thus We have

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where $\text{UNIFY}(\ell_i, \neg m_j) = \theta$. For example, we can resolve the two clauses

$$[Animal(F(x)) \vee Loves(G(x), x)] \quad \text{and} \quad [\neg Loves(u, v) \vee \neg Kills(u, v)]$$

by eliminating the complementary literals $Loves(G(x), x)$ and $\neg Loves(u, v)$, with unifier $\theta = \{u/G(x), v/x\}$, to produce the **resolvent** clause

$$[Animal(F(x)) \vee \neg Kills(G(x), x)] \ .$$

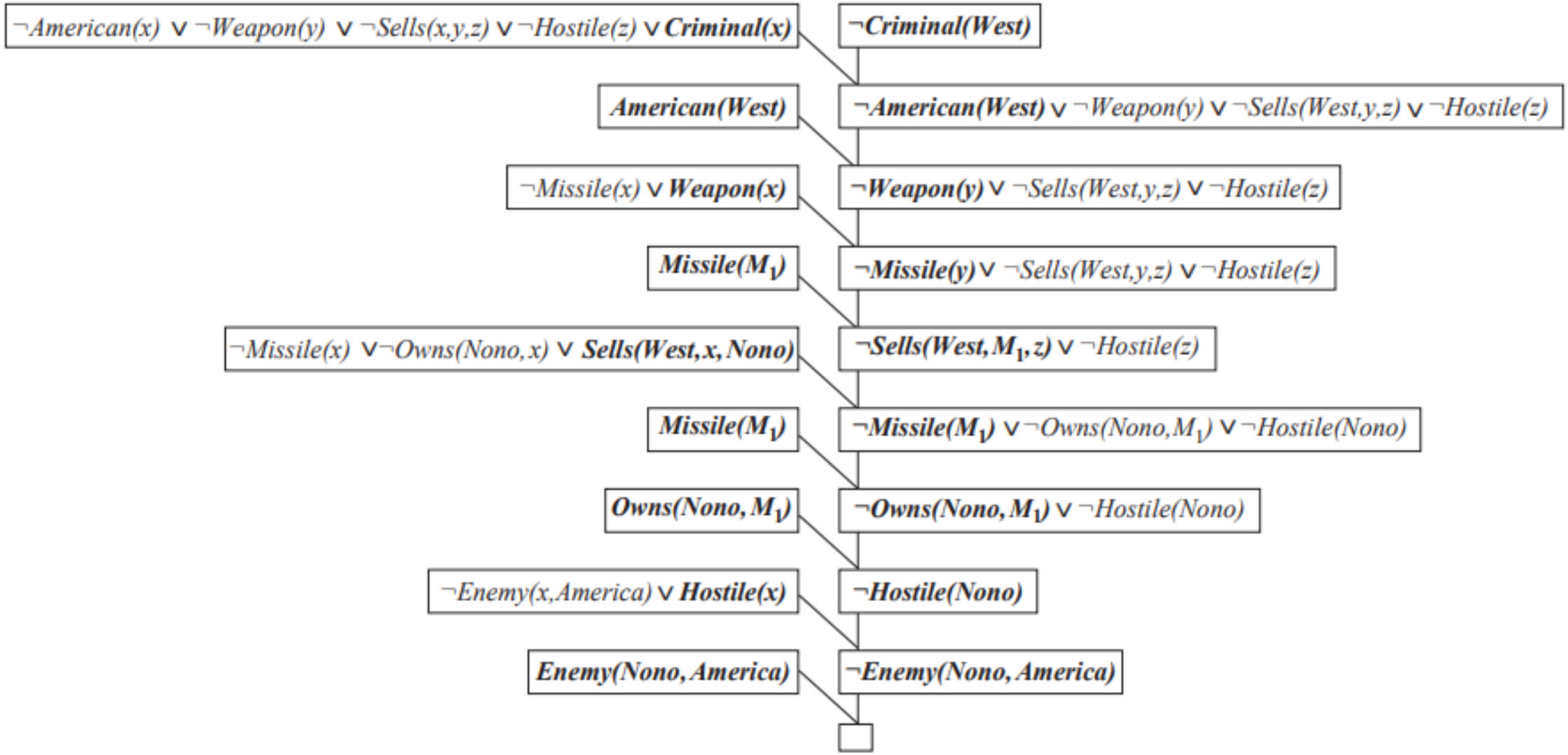


**Figure 9.11** A resolution proof that West is a criminal. At each step, the literals that unify are in bold.

# Another Example

- Everyone who loves all animals is loved by someone.
- Anyone who kills an animal is loved by no one.
- Jack loves all animals.
- Either Jack or Curiosity killed the cat, who is named Tuna.
- Did Curiosity kill the cat?

First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

A. $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)]$

B. $\forall x\ [\exists z\ Animal(z) \wedge Kills(x,z)] \Rightarrow [\forall y\ \neg Loves(y,x)]$

C. $\forall x\ Animal(x) \Rightarrow Loves(Jack,x)$

D. $Kills(Jack,Tuna) \vee Kills(Curiosity,Tuna)$

E. $Cat(Tuna)$

F. $\forall x\ Cat(x) \Rightarrow Animal(x)$

¬G. $\neg Kills(Curiosity,Tuna)$

Now we apply the conversion procedure to convert each sentence to CNF:

A1. $Animal(F(x)) \vee Loves(G(x),x)$

A2. $\neg Loves(x,F(x)) \vee Loves(G(x),x)$

B. $\neg Loves(y,x) \vee \neg Animal(z) \vee \neg Kills(x,z)$

C. $\neg Animal(x) \vee Loves(Jack,x)$

D. $Kills(Jack,Tuna) \vee Kills(Curiosity,Tuna)$

E. $Cat(Tuna)$

F. $\neg Cat(x) \vee Animal(x)$

¬G. $\neg Kills(Curiosity,Tuna)$

Suppose Curiosity did not kill Tuna. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore, Curiosity killed the cat.

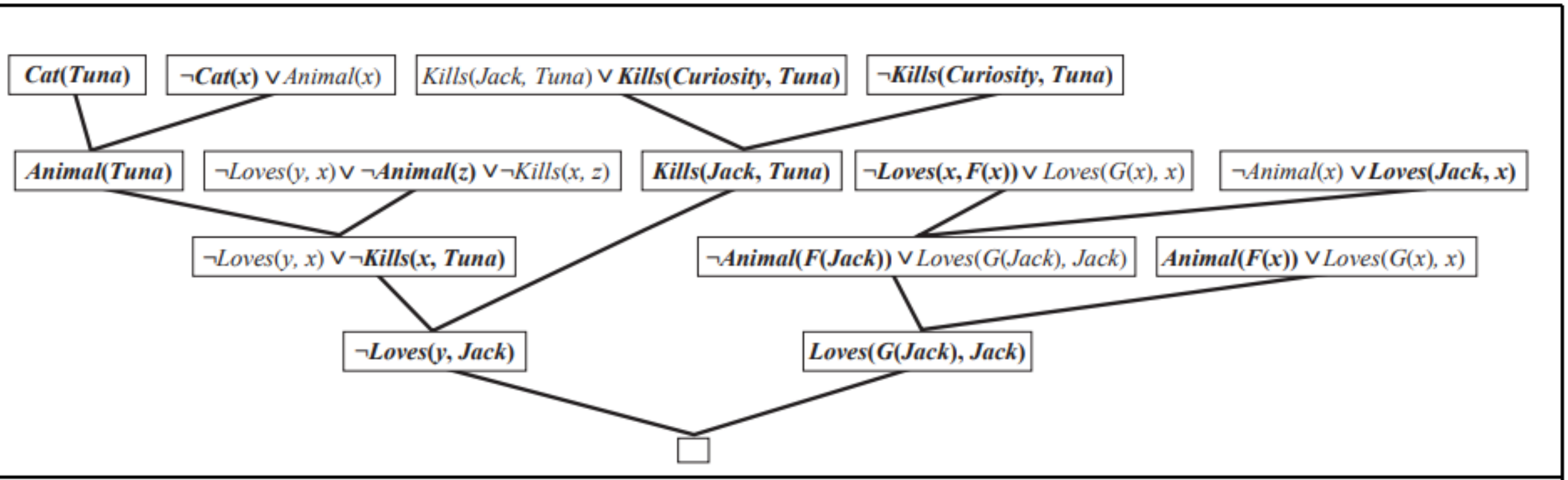


**Figure 9.12** A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $Loves(G(Jack), Jack)$. Notice also in the upper right, the unification of $Loves(x, F(x))$ and $Loves(Jack, x)$ can only succeed after the variables have been standardized apart.

# Completeness of resolution

- REFUTATION COMPLETENESS :Means that if a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction
- It can be used to find all answers to a given question, Q(x), by proving that KB $\wedge$ $\neg$Q(x) is unsatisfiable.
- If S is an unsatisfiable set of clauses, then the application of a finite number of resolution steps to S will yield a contradiction

# Proof of Completeness of resolution

The basic structure of the proof is as follows:

1. First, we observe that if S is unsatisfiable, then there exists a particular set of ground instances of the clauses of S such that this set is also unsatisfiable **(Herbrand's theorem).**
2. We then appeal to the **ground resolution theorem** given, which states that **propositional resolution is complete for ground sentences**.
3. We then use a **lifting lemma to show** that, for any propositional resolution proof using the set of ground sentences, **there is a corresponding first-order resolution proof** using the first-order sentences from which the ground sentences were obtained.

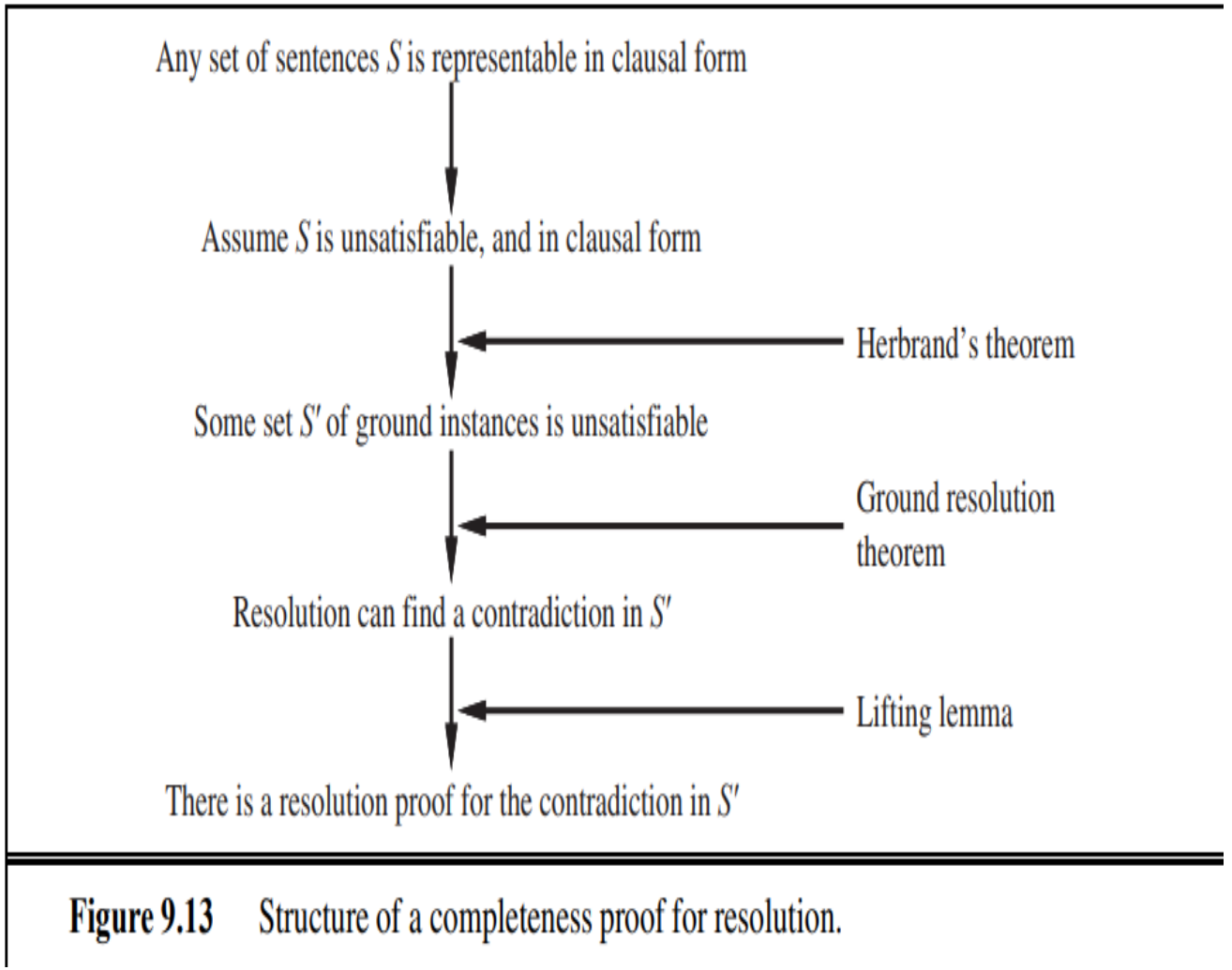


**Figure 9.13** Structure of a completeness proof for resolution.

# Lifting Lemma

Let C1 and C2 be two clauses with no shared variables, and let C ′ 1 and C ′ 2 be ground instances of C1 and C2. If C ′ is a resolvent of C ′ 1 and C ′ 2 , then there exists a clause C such that
(1) C is a resolvent of C1 and C2 and
(2) C ′ is a ground instance of C.

This is called a lifting lemma, because it lifts a proof step from ground clauses up to general first-order clauses

# Questions

- Discuss Completeness of  Resolution using the following concepts:
  - Herbrand universe
  - Saturation
  - Herbrand base
  - Lifting Lemma

# Equality

- Equality is reflexive, symmetric, and transitive,
- We can substitute equals for equals in any predicate or function
- We need three basic axioms, and then one for each predicate and function:

$$\forall x \;\; x = x$$
$$\forall x, y \;\; x = y \;\Rightarrow\; y = x$$
$$\forall x, y, z \;\; x = y \wedge y = z \;\Rightarrow\; x = z$$

$$\forall x, y \;\; x = y \;\Rightarrow\; (P_1(x) \;\Leftrightarrow\; P_1(y))$$
$$\forall x, y \;\; x = y \;\Rightarrow\; (P_2(x) \;\Leftrightarrow\; P_2(y))$$
$$\vdots$$
$$\forall w, x, y, z \;\; w = y \wedge x = z \;\Rightarrow\; (F_1(w, x) = F_1(y, z))$$
$$\forall w, x, y, z \;\; w = y \wedge x = z \;\Rightarrow\; (F_2(w, x) = F_2(y, z))$$
$$\vdots$$

# Equality

- **Demodulation** : The simplest rule, demodulation, takes a unit clause x = y and some clause α that contains the term x, and yields a new clause formed by substituting y for x within α.
- That means that demodulation can be used for simplifying expressions using demodulators such as
  - x + 0= x or $x^1$= x.
- As another example, given
  - Father (Father (x)) = PaternalGrandfather (x)
  - Birthdate(Father (Father (Bella)), 1926)
- we can conclude by demodulation
  - Birthdate(PaternalGrandfather (Bella), 1926) .

- **Demodulation**: For any terms $x$, $y$, and $z$, where $z$ appears somewhere in literal $m_i$ and where $\text{UNIFY}(x, z) = \theta$,

$$\frac{x = y, \qquad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), m_1 \vee \cdots \vee m_n)} .$$

  where SUBST is the usual substitution of a binding list, and $\text{SUB}(x, y, m)$ means to replace $x$ with $y$ everywhere that $x$ occurs within $m$.

The rule can also be extended to handle non-unit clauses in which an equality literal appears:

- **Paramodulation**: For any terms $x$, $y$, and $z$, where $z$ appears somewhere in literal $m_i$, and where $\text{UNIFY}(x, z) = \theta$,

$$\frac{\ell_1 \vee \cdots \vee \ell_k \vee x = y, \qquad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), \text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_n))} .$$

For example, from

**P(F(x, B), x) V Q(x) and F(A, y)= y V R(y)** we have

**θ = UNIFY(F(A, y), F(x, B))= {x/A, y/B},**

and we can conclude by paramodulation the sentence

**P(B, A) V Q(A) V R(B) .**

# **Demodulation**:

- Demodulation is the simplest rule used to handle equality in logic.
- It involves taking a unit clause of the form "x = y" and a clause α containing the term "x."
- It yields a new clause by substituting "y" for "x" within α, under the condition that the term within α unifies with "x" (i.e., they can be matched), but it doesn't necessarily have to be exactly equal to "x."
- This process is directional, meaning that given "x = y," "x" always gets replaced with "y," never vice versa.
- Demodulation can be used to simplify expressions using demodulators such as "x + 0 = x" or "x^1 = x."

# **Paramodulation**:

- Paramodulation extends demodulation to handle non-unit clauses where an equality literal appears.
- It involves terms "x," "y," and "z," where "z" appears somewhere in a literal "mi."
- Similar to demodulation, it requires the unification of "x" and "z" resulting in a substitution θ.
- With paramodulation, instead of just handling unit clauses, it can deal with clauses containing multiple literals.
- The process substitutes "y" for "x" within the clause, while also applying the substitution θ to all other literals in the clause where "x" appears.

# Summary

1. **Unification:** *is a process used to find a common instantiation for two predicates or terms such that they become identical.*
2. **Forward Chaining:** *is a reasoning and inference procedure which starts with known facts and moves forward to reach conclusions*
3. **Backward Chaining:** *is a reasoning and inference procedure which starts with the goal and moves backward to verify if the goal can be satisfied,*
4. **Resolution:** *is an inference rule used to derive new clauses by combining existing ones***.**

These ***techniques are essential for reasoning and inference in First-Order Logic systems***.