Unit 7 Knowledge Representation Using Logic- II

Structure:

7.1 Introduction

In the last unit you have learnt how to represent simple facts in logic and the role of logic in Artificial Intelligence. You have also learnt the syntax and semantics for propositional logic. In this unit you will be introduced to Inference rules for propositional logic. You will also study conversion to clausal form and resolution principle and its algorithm.

Objectives:

After studying this unit, you should be able to

- explain the Inference Rules for Propositional Logic
- list and explain the steps to transform a sentence into Clausal Form
- describe Resolution Principle
- explain Resolution Algorithm
- list and explain Resolution strategies.

7.2 Inference Rules for Propositional Logic

The process by which the soundness of an inference rule is established through truth tables can be extended to entire classes of inferences. There are certain patterns of inferences that occur over and over, and their soundness can be shown once and for all. Then the pattern can be captured in what is called an **inference rule.** Once this has been done, the rule can be used to make inferences without going through the tedious process of building truth tables.

The notation $\alpha \rightarrow \beta$ says that β can be derived from α by inference. An alternate notation emphasizes that this is not a sentence, but rather an inference rule:

$$\frac{\alpha}{\beta}$$

Here α and β match sentences.

Whenever something in the knowledge base matches the pattern above the line, the inference rule concludes the sentence below the line. An inference rule is sound if the conclusion is true in all cases in which the premises are true.

Some example inference rules:

Modus Ponens

From P and P → Q infer Q. This is sometime written as

$$\frac{\begin{array}{c} P \\ P \rightarrow Q \end{array}}{Q}$$

Foe example

given: (Amit is a father)
and: (Amit is a father) → (Amit has a child)
conclude: (Amit has a child)

Chain Rule

From P → Q, and Q → R, infer P → R. Or

$$\frac{\begin{array}{c} P \rightarrow Q \\ Q \rightarrow R \end{array}}{P \rightarrow R}$$

For example,

given: (programmers like LISP) → (programmers hate COBOL)
and: (programmers hate COBOL) → (programmers like recursion)
conclude: (programmers like LISP) → (programmers like recursion)

Substitution

If s is a valid sentence, s' derived from s by consistent substitution of propositions in s, is also valid. For example, the sentence P ∨ ˜P is valid; therefore Q ∨ ˜Q is also valid by the substitution rule.

Simplification

From P & Q infer P.

Conjunction

From P and from Q, infer P & Q.

Transportation

From P → Q, infer ˜Q → ˜P.

Self Assessment Questions

1. The notation α → β says that β can be derived from α by __________.
2. An inference rule is sound if the conclusion is true in all cases in which the __________ are true.

7.3 Conversion to Clausal Form

We define a *clause* as the disjunction of a number of literals. A *ground clause* is one in which no variables occur in the expression. A *Horn clause* is a clause with at the most, one positive literal. One method called *Resolution* requires that all statements be converted into normalized clausal form.

Steps to transform a sentence into clausal form:

- Eliminate implications
- Move ˜ inwards
- Standardize variables
- Skolemize
- Move quantifiers left
- Distribute ∨ over ∧
- Flatten nested conjunctions and disjunctions
- Convert disjunctions to implications

Step 1: Eliminate Implications

Eliminate all implication and equivalency connectives (use ˜P ∨ Q in place of P→ Q and (˜P ∨ Q) ∧ (˜Q ∨ P) in place of P ↔ Q.

Step 2: Move ˜ inwards

Move all negations in, to immediately precede an atom (use P in place of ˜(˜P), and DeMorgan' laws, ∃ x ˜F[x] in place of ˜(x) F[x] and x ˜ F[x] in place of ˜(∃ x) F[x].

Or in other words

˜(P ∨ Q)	becomes	˜ P ∧ ˜ Q
˜(P ∧ Q)	becomes	˜ P ∨ ˜ Q
˜∀ x,P	becomes	∃ x ˜P
˜∃ x,P	becomes	∀x ˜P
˜(˜P)	becomes	P

Step 3: Standardize variables

Rename variables, if necessary, so that all quantifiers have different variables assignments; that is, rename variables so that variables bound by one quantifier are not the same as variables bound by a different quantifier. For example, in the expression ∀ x (P(x) → (∃ x (Q(x))) rename the second dummy variable x which is bound by the existential quantifier to be a different variable, say y, to give ∀ x (P(x) → (∃y (Q(y))). This avoids the confusion later when we drop the quantifiers.

Step 4: Skolemize

Skolemization is the process of removing existential quantifiers by elimination. In the simple case translate ∃ x P(x) into P(A), where A is a constant that does not appear elsewhere in the KB. But there is the added complication that some of the existential quantifiers, even though move left, may still be nested inside a universal quantifier.

Skolemization is accomplished as follows:

i) If the first (leftmost) quantifier in an expression is an existential quantifier, replace all occurrences of the variable it quantifies with an arbitrary constant not appearing elsewhere in the expression and delete the quantifier. The same procedure should be followed for all other existential quantifiers not preceded by a universal quantifier, in each case, using different constant symbols in the substitution.

ii) For each existential quantifier that is preceded by one or more universal quantifiers (is within the scope of one or more universal quantifiers), replace all occurrences of the existentially quantified variable by a function symbol not appearing elsewhere in the expression. The argument assigned to the function should match all the variables appearing in each universal quantifier which precedes the existential quantifier. This existential quantifier should then be deleted. The same procedure should be repeated for each remaining existential quantifier using a different function symbol and choosing function arguments that correspond to all universally quantified variables that precede the existentially quantified variable being replaced.

Example of Skolemization (1):

Consider "Everyone has a heart":

$$\forall x \ Person(x) \rightarrow \exists \ y \ Heart(y) \wedge Has(x,y)$$

If we just replaced y with a constant, H, we would get

$$\forall x \ Person(x) \rightarrow Heart(H) \wedge Has(x,H)$$

which says that everyone has the same heart H. We need to say that the heart they have is not necessarily shared, that is, it can be found by applying to each person a function that maps from person to heart:

$$\forall x \ Person(x) \rightarrow Heart(F(x)) \wedge Has(x,F(x))$$

where F is a function name that does not appear elsewhere in the KB. F is called a **Skolem function**. In general, the existentially quantified variable is replaced by a term that consists of a Skolem function applied to all the variables universally quantified outside the existential quantifier in question. Skolemization eliminates all existentially quantified variables, so we are now free to drop the universal quantifiers, because any variable must be universally quantified.

Example of Skolemization (2):

Given the expression

$$\exists u \ \forall v \ \forall x \ \exists \ y \ P(F(u), v, x, y) \rightarrow Q(u,v,y)$$

The Skolem form is determined as

$$\forall v \ \forall x \ P(F(a), v, x, g(v,x)) \rightarrow Q(a, v, g(v,x))$$

In making the substitution, it should be noted that the variable u appearing after the first existential quantifier has been replaced in the second expression by the arbitrary constant a. The constant did not appear elsewhere in the first expression. The variable y has been replaced by the function symbol g having the variable v and x as arguments, since both of these variables are universally quantified to the left of the existential quantifier for y. Replacement of y by an arbitrary function with arguments v and x is justified on the basis that y, following v and x, may be functionally dependent on them and, if so, the arbitrary function g can account for this dependency.

Step 5: Move quantifiers left

Move all universal quantifiers to the left of the expression and put the expression on the right into CNF.

Step 6: Distribute $\vee$ over $\wedge$

$(a \wedge b) \vee c$ becomes $(a \vee c) \wedge (b \vee c)$

Step 7: Flatten nested conjunctions and disjunctions

$(a \wedge b) \wedge c$ becomes $(a \wedge b \wedge c)$ and

$(a \vee b) \vee c$ becomes $(a \vee b \vee c)$

Step 8: Convert disjunctions to implications

Optionally, you can take one more step to convert to implicative normal form. For each conjunct, gather up the negative literals into one list, the positive literals into another list, and build an implication from them:

$(\tilde{}a \vee \tilde{}b \vee c \vee d)$ becomes $(a \wedge b \rightarrow c \vee d)$

Example: Conversion to Clausal Form

Let us convert the expression

$\exists x\ \forall y\ (\forall z\ P(F(x), y, z) \rightarrow (\exists u\ Q(x, u) \wedge \exists\ v\ R(y, v)))$

in clausal form.

Solution:

After application of step 1, we obtain

$\exists x\ \forall y\ (\tilde{}(\forall z)\ P(F(x), y, z) \vee (\exists u\ Q(x, u) \wedge (\exists\ v)\ R(y, v))))$

After application of step 2, we obtain

$\exists x\ \forall y\ (\exists z\ \tilde{}P(F(x), y, z) \vee (\exists u\ Q(x, u) \wedge (\exists v)\ R(y, v))))$

Step 3 is not required.

After application of step 4, we obtain

$\forall y\ (\tilde{}P(F(a), y, g(y)) \vee (Q(a, h(y)) \wedge R(y, l(y)))$

After application of step 5, we obtain

$$\forall y\ ((\tilde{}P(F(a), y, g(y)) \vee Q(a, h(y)) \wedge (\tilde{}P(F(a), y, g(y)) \vee R(y, l(y)))$$

Finally, after application of remaining steps, we obtain the clausal form

$\tilde{}P(F(a), y, g(y)) \vee Q(a, h(y))$

$\tilde{}P(F(a), y, g(y)) \vee R(y, l(y))$

Self Assessment Questions

3. A ________ clause is one in which no variables occur in the expression.
4. _______ is the process of removing existential quantifiers by elimination.

7.4 Resolution

Resolution is a syntactic inference procedure which, when applied to a set of clauses, determines if the set is unsatisfiable. This process is similar to the process of obtaining a proof by contradiction. For example, suppose we have the set of clauses (axioms) $C_1, C_2, \ldots, C_n$ and we wish to deduce or prove the clause D, that is, to show that D is a logical sequence of C_1 & C_2 & … & C_n. First we negate D and add ˜D to the set of clauses $C_1, C_2, \ldots, C_n$. Then, using resolution together with factoring, we can show that the set is unsatisfiable by deducing a contradiction. Such a proof is called a proof by refutation which, if successful, yields the empty clause denoted by []. The empty clause is always false since no interpretation can satisfy it. It is derived from combining contradictory clauses such as P and ˜P. Resolution with factoring is *complete* in the sense that it will always generate the empty clause from a set of unsatisfiable clauses.

Resolution is very simple. Given two clauses C_1 and C_2 with no variables in common, if there is a literal l_1 in C_1 which is complement of a literal l_2 in C_2, both l_1 and l_2 are deleted and a disjuncted C is formed from the remaining reduced clauses. The new clause C is called the *resolvent* of C_1 and C_2. Resolution is the process of generating these resolvents from a set of clauses. For example, to resolve the two clauses

$(\tilde{}P \vee Q)$ and $(\tilde{}Q \vee R)$

We write

$$\frac{\tilde{}P \vee Q,\ \tilde{}Q \vee R}{\tilde{}P \vee R}$$

7.4.1 Resolution algorithm

Resolution algorithm is as follows:

1) Choose two clauses that have *exactly one* pair of literals that are complementary (have different signs).
2) Produce a new clause by deleting the complementary literals and combining the remaining literals.
3) If the resulting clause is empty ("box"), stop; the theorem is proved by contradiction. (If the negation of the theorem leads to a contradiction, then the theorem must be true.)

This assumes that the premises are consistent.

7.4.2 Resolution strategy

Many different strategies have been tried for selecting the clauses to be resolved. These include:

- **Level saturation or two-pointer method**
  The outer pointer starts at the negated conclusion; the inner pointer starts at the first clause. The two clauses denoted by the pointers are resolved if possible, with the result added to the end of the list of clauses. (There may be multiple resolvents of two clauses.) The inner pointer is incremented to the next clause until it reaches the outer pointer; then the outer pointer is incremented and the inner pointer is reset to the front. The two-pointer method is a breadth-first method that will generate many duplicate clauses.
- **Set of support**
  One clause in each resolution step must be part of the negated conclusion or a clause derived from it. This can be combined with the two-pointer method by putting the clauses from the negated conclusion at the end of the list. Set-of-support keeps the proof process focused on the theorem to be proved rather than trying to prove everything.
- **Unit preference**
  Clauses are prioritized, with *unit clauses* (those with only one literal) preferred, or more generally, shorter clauses preferred. Resolution with a unit clause makes the result smaller.
- **Linear resolution**
  One clause in each step must be the result of the previous step. This is a depth-first strategy. It may be necessary to back up to a previous clause if no resolution with the current clause is possible.

7.4.3 Unification algorithm

In propositional logic, it is easy to determine that two literals can not both be true at the same time. Simply look for L and ˜L. In predicate logic, the matching process is more complicated since the arguments of the predicates must be considered. For example, man (John) and ˜man(John) is a contradiction, while man (John) and ˜ man(Amit) is not. Thus in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. There is a straightforward recursion procedure, called the *unification algorithm,* that does just this.

The basic unification algorithm is simple. However, it must be implemented with care to ensure that the results are correct.

We begin by making sure that the two expressions have no variables in common. If there are common variables, substitute a new variable in one of the expressions. (Since variables are universally quantified, another variable can be substituted without changing the meaning.)

Imagine moving a pointer left-to-right across both expressions until parts are encountered that are not the same in both expressions. If one is a *variable*, and the other is a *term not containing that variable*,

1) Substitute the term for the variable in both expressions
2) Substitute the term for the variable in the existing substitution set (This is necessary so that the substitution set will be simultaneous.)
3) Add the substitution to the substitution set.

Example of Unification (1):

Suppose we want to unify the expressions

P(a,a)
P(b,c)

The two instance of P match fine. Next we compare a and b, and decide that if we substitute b for a, they could match. We will write that substitution as

b/a

We could, of course, have decided instead to substitute a for b, since they are both just dummy variable names. The algorithm will simply pick one of these two substitutions. But now, if we simply continue and match a and c, we produce the substitution c/a. but we can not substitute both b and c for a, so we have to proceed a consistent substitution.

What we need to do after finding the first substitution b/a is to make that substitution throughout the literals, giving

P(b,b)
P(b,c)

Now we can attempt to unify arguments b and c, which succeeds the substitution c/b. the entire unification process has now succeeded a substitution that is the composition of the two substitutions we found. We write the composition as

(c/b)(b/a)

following standard notation for function composition. In general, the substitution (a1/a2,a3/a4, …)(b1/b2,b3/b4, …) … means to apply all the substitutions of the rightmost list, then take the result and apply all the ones of the next list, and so forth, until all substitutions have been applied.

The object of unification process is to discover at least one substitution that causes two literals to match. Usually, if there is one such substitution there are many. For example, the literals

care(a,b)
care(Amit,c)

could be unified with any of the following substitutions:

(Amit/a, c/b)
(Amit/a, b/c)
(Amit/a, Akash/b, Akash/c)
(Amit/a, Arvind/b, Arvind/c)

The first two of these are equivalent except for lexical variation. But the second two, although they produce a match, also produce a substitution that is more restrictive than absolutely necessary for the match. Because the final substitution produced by the unification process will be used by the resolution procedure, it is useful to generate the most general unifier possible. The algorithm shown next will do that.

Example of Unification (2):

Consider unifying the literal *P(x, g(x))* with:

1) *P(z,y)* : unifies with *{x/z, g(x)/y}*
2) *P(z, g(z))*: unifies with *{x/z}* or *{z/x}*

3) *P(Socrates, g(Socrates))* : unifies, *{Socrates/x}*
4) *P(z, g(y))*: unifies with *{x/z, x/y}* or *{z/x, z/y}*
5) *P(g(y), z)*: unifies with *{g(y)/x, g(g(y))/z}*
6) *P(Socrates, f(Socrates))* : does not unify: *f* and *g* do not match.
7) *P(g(y), y)* : does not unify: no substitution works.

A procedure is described to Unify (L1,L2), which returns as its value a list representing the composition of the substitutions that were performed during the match. The empty list, NIL, indicates that a match was found without any substitutions. The list consisting of the single value FAIL indicates that the unification procedure failed.

Algorithm: Unify(L1,L2):

```
1) If L1 or L2 are both variables or constants, then:
   a) If L1 and L2 are identical, then return NIL.
   b) Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL},
      else return (L2/L1).
   c) Else if L2 is a variable, then if L2 occurs in L1 then return {FAIL},
      else return (L1/L2).
   d) Else return {FAIL}.
2) If the initial predicate symbols in L1 and L2 are not identical, then return
   {FAIL}.
3) If L1 and L2 have a different number of arguments, then return {FAIL}.
4) Set SUBST to NIL. (At the end of this procedure, SUBST will contain all
   the substitutions used to unify L1 and L2.)
5) For i←1 to the number of arguments in L1:
   a) Call Unify with the $i^{th}$ argument of L1 and the $i^{th}$ argument of L2,
      putting result in S.
   b) If S contains FAIL then return {FAIL}.
   c) If S is not equal to NIL then:
      i) Apply S to the remainder of both L1 and L2.
      ii) SUBST := APPEND(S,SUBST).
6) Return SUBST.
```

7.4.4 Example of Resolution: Monkey and banana problem

We envision a room containing a monkey, a chair, and some bananas that have been hung from the center of the ceiling, out of reach from the monkey. If the monkey is clever enough, he can reach the bananas by placing the chair directly below them and climbing on top of the chair. The problem is to prove the monkey can reach the bananas using resolution.

In creating a knowledge base, it is essential first to identify all relevant objects which will play some role in the anticipated inferences. Where possible, irrelevant objects should be omitted, but never at the risk of incompleteness. For example, in the current problem, the monkey, bananas, and the chair are essential. Also needed is some reference object such as the floor or ceiling to establish the height relationship between monkey and bananas. Other objects such as windows, walls or doors are not relevant.

The next step is to establish important properties of objects, relation between them, and any assertions likely to be needed. These include such facts as the chair is tall enough to raise the monkey within the reach of the bananas, the monkey is dexterous, the chair can be moved under the bananas, and so on. Again, all important properties, relations, and assertions should be included and irrelevant ones omitted. Otherwise, unnecessary inference steps may be taken.

The important factors for our problem are described below, and all items needed for the actual knowledge base are listed as axioms. These are the essential facts and rules. Although not explicitly indicated, all variables are universally quantified.

Relevant factors for the problem

In this section, you will be introduced to the Important factors required for our problem.

CONSTANTS

{floor, chair, bananas, monkey}

VARIABLES

{x,y,z}

PREDICATES

{can_reach(x,y)	**; x can reach y**
dexterous(x)	**; x is a dexterous animal**
close(x,y)	**; x is close to y**
get_on(x,y)	**; x can get on y**
under(x,y)	**; x is under y**
tall(x)	**; x is tall**

in_room(x)	**; x is in the room**
can_move(x,y,z)	**; x can move y near z**
can_climb(x,y)}	**; x can climb onto y**

AXIOMS

{in_room(bananas)
in_room(chair)
in_room(monkey)
dexterous(monkey)
tall(chair)
close(bananas,floor)
can_move(monkey,chair,bananas)
can_climb(monkey,chair)
(dexterous(x) & close(x,y)) → can-reach(x,y)

((get_on(x,y) & under(y,bananas) & tall(y) → close(x,bananas))
((in_room(x) & in_room(y) & in_room(z) & can_move(x,y,z)) → close(z,floor) V under(y,z))
(can_climb(x,y) → get_on(x,y))}

Using the above axioms, a knowledge base can be written down directly in the required clausal form. All that is needed to make the necessary substitutions are the equivalences

P → Q = ˜P V Q

and De Morgan's laws.

Clausal form of knowledge base

1) **{in_room(monkey)**
2) **in_room(bananas)**
3) **in_room(chair)**
4) **tall(chair)**
5) **dexterous(monkey)**
6) **can_move(monkey,chair,bananas)**
7) **can_climb(monkey,chair)**
8) **˜close(bananas,floor)**
9) **˜can_climb(x,y) $^\vee$ get_on(x,y)**
10) **˜dexterous(x) $^\vee$ ˜close(x,y) $^\vee$ can_reach(x,y)**
11) **˜get_on(x,y) $^\vee$ ˜under(y,bananas) $^\vee$ ˜tall(y) $^\vee$ close(x,bananas)**

12) **˜in_room(x) ∨ ˜in_room(y) ∨ ˜in_room(z) ∨ ˜can_move(x,y,z) ∨ close(y,floor) ∨ under(y,z)**

13) **can_reach(monkey,bananas)**

Resolution Proof: A proof that the monkey can reach the bananas is summarized below. As we can see, this is a refutation proof where the statement to be proved (can_reach(monkey,bananas)) has been negated and added to the knowledgebase (number 13). The proof then follows when a contradiction is found (see number 23). In other words we can say that for proving that the monkey can reach the bananas, we shall be following the principle of minus-minus-plus.

14)	**˜can_move(monkey,chair,bananas) ∨ close(bananas,floor) ∨ under(chair,bananas)**	**:** 14 is a resolvent of 1,2,3 and 12 with substitution {monkey/x, chair/y, bananas/z}
15)	**close(bananas,floor) ∨ under(chair,bananas)**	; 15 is a resolvent of 6 and
16)	**under(chair,bananas)**	; 16 is a resolvent of 8 and
17)	**get_on(x,chair) ∨ ˜tall(chair) ∨ Close(x,bananas)**	; 17 is a resolvent of 11 and 16 with substitution {chair/y}
18)	**˜get_on(x,chair) ∨ close(x,bananas)**	; 18 is a resolvent of 4 and17
19**)**	**get_on(monkey,chair)**	; 19 is a resolvent of 7 and 9
20)	**close(monkey,bananas)**	; 20 is a resolvent of 18 and 19 with substitution {monkey/x}
21)	**˜close(monkey,y) ∨ can_reach(monkey,y)**	; 21 is a resolvent of 10 and 5 with substitution {monkey/x}
22)	**reach(monkey,bananas)**	; 20 is a resolvent of 20 and 21 with substitution {bananas/y}
23)	[ ]	: 23 is a resolvent of 13 and 22

In performing the above proof, no particular strategy was followed. Clearly, however, good choices were made in selecting parent clause for resolution.

Otherwise, many unnecessary steps may have been taken before completing the proof. Different forms of resolution were completed in steps 14 through 23.

Self Assessment Questions

5. ____________ is a syntactic inference procedure which, when applied to a set of clauses, determines if the set is unsatisfiable.
6. __________ algorithm is a straightforward recursion procedure.
7. In creating a _____________, it is essential first to identify all relevant objects which will play some role in the anticipated inferences.
8. In propositional logic, it is easy to determine that two literals cannot both be true at the same time.(State True or False)

7.5 Summary

In this unit you have learnt the inference rules for propositional logic. We have also discussed steps to transform a sentence into Clausal Form and resolution principle, its algorithm and strategy. The process by which the soundness of an inference rule is established through truth tables can be extended to entire classes of inferences. The notation $\alpha \rightarrow \beta$ says that β can be derived from α by inference. . A Horn clause is a clause with at the most, one positive literal. Resolution is a syntactic inference procedure which, when applied to a set of clauses, determines if the set is unsatisfiable. In order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical.

7.6 Terminal Questions

1. Write a note on inference rules for propositional logic
2. What are the steps to transform a sentence into clausal form?
3. What do you mean by resolution?
4. Explain the resolution strategies.
5. Write a note on unification algorithm.

7.7 Answers

Self Assessment Questions

1. Inference.
2. Premises

3. Ground
4. Skolemization
5. Resolution
6. Unification
7. Knowledge base
8. True

Terminal Questions

1. Certain patterns of inferences occur over and over, and their soundness can be shown once and for all. Then the pattern can be captured in what is called an inference rule. (Refer section 7.2 for detail)
2. All the eight steps starting from 'eliminating implications' and ending with 'converting disjunctions to implications' are explained in section 7.3. Refer the same for details.
3. Resolution is a syntactic inference procedure. (Refer section 7.4 for detail)
4. Many different strategies have been tried for selecting the clauses to be resolved. These include:
   - Level saturation or two-pointer method

   (Refer sub-section 7.4.2 for detail)
5. In order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. There is a straightforward recursion procedure, called the unification algorithm that does just this.
   (Refer sub-section 7.4.3 for detail)