UNIT-IV

USING PREDICATE LOGIC

Representation of Simple Facts in Logic

Propositional logic is useful because it is simple to deal with and a decision procedure for it exists.

Also, In order to draw conclusions, facts are represented in a more convenient way as,

1. Marcus is a man.
    - man(Marcus)
2. Plato is a man.
    - man(Plato)
3. All men are mortal.
    - mortal(men)

But propositional logic fails to capture the relationship between an individual being a man and that individual being a mortal.

- How can these sentences be represented so that we can infer the third sentence from the first two?
- Also, Propositional logic commits only to the existence of facts that may or may not be the case in the world being represented.
- Moreover, It has a simple syntax and simple semantics. It suffices to illustrate the process of inference.
- Propositional logic quickly becomes impractical, even for very small worlds.

Predicate logic

First-order Predicate logic (FOPL) models the world in terms of

- Objects, which are things with individual identities
- Properties of objects that distinguish them from other objects
- Relations that hold among sets of objects

- Functions, which are a subset of relations where there is only one "value" for any given "input"

First-order Predicate logic (FOPL) provides

- Constants: a, b, dog33. Name a specific object.
- Variables: X, Y. Refer to an object without naming it.
- Functions: Mapping from objects to objects.
- Terms: Refer to objects
- Atomic Sentences: in(dad-of(X), food6) Can be true or false, Correspond to propositional symbols P, Q.

A well-formed formula (*wff*) is a sentence containing no "free" variables. So, That is, all variables are "bound" by universal or existential quantifiers.

$(\forall x)P(x, y)$ has x bound as a universally quantified variable, but y is free.

Quantifiers

Universal quantification

- $(\forall x)P(x)$ means that P holds for all values of x in the domain associated with that variable
- E.g., $(\forall x)$ dolphin(x) $\rightarrow$ mammal(x)

Existential quantification

- $(\exists x)P(x)$ means that P holds for some value of x in the domain associated with that variable
- E.g., $(\exists x)$ mammal(x) $\wedge$ lays-eggs(x)

Also, Consider the following example that shows the use of predicate logic as a way of representing knowledge.

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. Also, All Pompeians were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

The facts described by these sentences can be represented as a set of well-formed formulas (*wffs*) as follows:

1. Marcus was a man.
    - man(Marcus)
2. Marcus was a Pompeian.
    - Pompeian(Marcus)
3. All Pompeians were Romans.
    - $\forall x$: Pompeian(x) $\rightarrow$ Roman(x)
4. Caesar was a ruler.
    - ruler(Caesar)
5. All Pompeians were either loyal to Caesar or hated him.
    - inclusive-or
    - $\forall x$: Roman(x) $\rightarrow$ loyalto(x, Caesar) $\vee$ hate(x, Caesar)
    - exclusive-or
    - $\forall x$: Roman(x) $\rightarrow$ (loyalto(x, Caesar) $\wedge\neg$ hate(x, Caesar)) $\vee$
    - ($\neg$loyalto(x, Caesar) $\wedge$ hate(x, Caesar))

6. Everyone is loyal to someone.
    - ∀x: ∃y: loyalto(x, y)
7. People only try to assassinate rulers they are not loyal to.
    - ∀x: ∀y: person(x) ∧ ruler(y) ∧ tryassassinate(x, y)
    - →¬loyalto(x, y)
8. Marcus tried to assassinate Caesar.
    - tryassassinate(Marcus, Caesar)

Now suppose if we want to use these statements to answer the question: ***Was Marcus loyal to Caesar?***

Also, Now let's try to produce a formal proof, reasoning backward from the desired goal: ¬ Ioyalto(Marcus, Caesar)

In order to prove the goal, we need to use the rules of inference to transform it into another goal (or possibly a set of goals) that can, in turn, transformed, and so on, until there are no unsatisfied goals remaining.

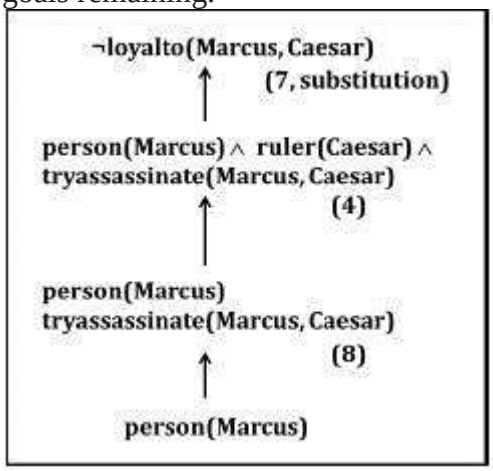

Figure: An attempt to prove ¬loyalto(Marcus, Caesar).

- The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. Also, We need to add the representation of another fact to our system, namely***: ∀man(x) → person(x)***
- Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.
- Moreover, From this simple example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones:
    1. Many English sentences are ambiguous (for example, 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
    2. Also, There is often a choice of how to represent the knowledge. Simple representations are desirable, but they may exclude certain kinds of reasoning.
    3. Similalry, Even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand. In order to be able to use a set of statements effectively. Moreover, It is usually necessary to have access to another set of statements that represent facts that people consider too obvious to mention.

Representing Instance and ISA Relationships

- Specific attributes **instance** and **isa** play an important role particularly in a useful form of reasoning called property inheritance.
- The predicates instance and isa explicitly captured the relationships they used to express, namely class membership and class inclusion.
- 4.2 shows the first five sentences of the last section represented in logic in three different ways.
- The first part of the figure contains the representations we have already discussed. In these representations, class membership represented with unary predicates (such as Roman), each of which corresponds to a class.
- Asserting that P(x) is true is equivalent to asserting that x is an instance (or element) of P.
- The second part of the figure contains representations that use the ***instance*** predicate explicitly.

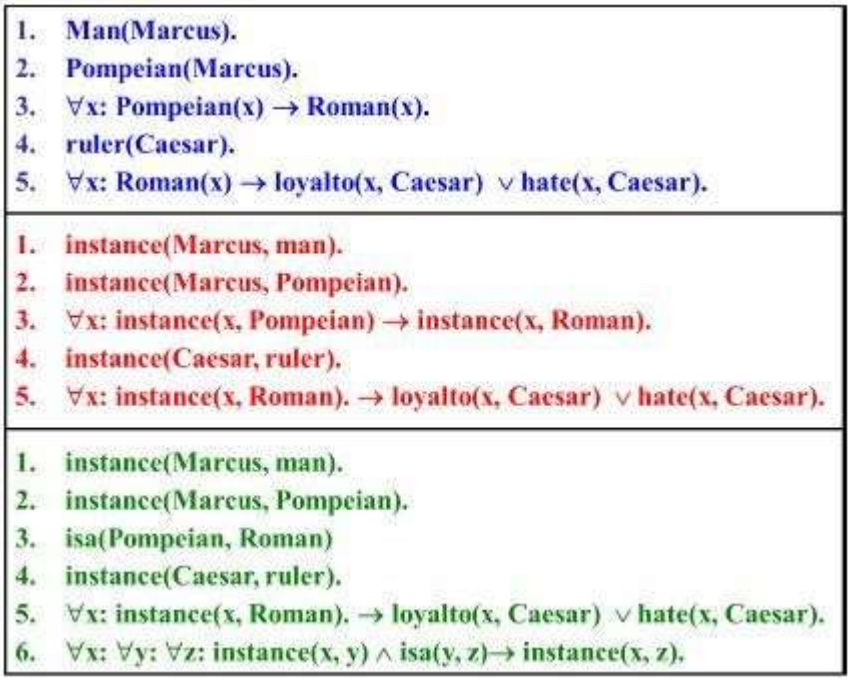

Figure: Three ways of representing class membership: ISA Relationships

- The predicate ***instance*** is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs.
- But these representations do not use an explicit ***isa*** predicate.
- Instead, subclass relationships, such as that between Pompeians and Romans, described as shown in sentence 3.
- The implication rule states that if an object is an instance of the subclass Pompeian then it is an instance of the superclass Roman.
- Note that this rule is equivalent to the standard set-theoretic definition of the subclass-superclass relationship.
- The third part contains representations that use both the ***instance*** and ***isa*** predicates explicitly.
- The use of the ***isa*** predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided.

Computable Functions and Predicates

- To express simple facts, such as the following greater-than and less-than relationships: gt(1,O) It(0,1) gt(2,1) It(1,2) gt(3,2) It(2,3)
- It is often also useful to have computable functions as well as computable predicates. Thus we might want to be able to evaluate the truth of gt(2 + 3,1)
- To do so requires that we first compute the value of the plus function given the arguments 2 and 3, and then send the arguments 5 and 1 to gt.

Consider the following set of facts, again involving Marcus:

1) Marcus was a man.
   man(Marcus)
2) Marcus was a Pompeian.
   Pompeian(Marcus)
3) Marcus was born in 40 A.D.
   born(Marcus, 40)
4) All men are mortal.
   x: man(x) → mortal(x)
5) All Pompeians died when the volcano erupted in 79 A.D.
   erupted(volcano, 79) ∧ ∀ x : [Pompeian(x) → died(x, 79)]
6) No mortal lives longer than 150 years.
   x: t1: At2: *mortal(x) born(x, t1) gt(t2 – t1,150) → died(x, t2)*
7) It is now 1991.
   now = 1991

So, Above example shows how these ideas of computable functions and predicates can be useful. It also makes use of the notion of equality and allows equal objects to be substituted for each other whenever it appears helpful to do so during a proof.

- So, Now suppose we want to answer the question "Is Marcus alive?"
- The statements suggested here, there may be two ways of deducing an answer.
- Either we can show that Marcus is dead because he was killed by the volcano or we can show that he must be dead because he would otherwise be more than 150 years old, which we know is not possible.
- Also, As soon as we attempt to follow either of those paths rigorously, however, we discover, just as we did in the last example, that we need some additional knowledge. For example, our statements talk about dying, but they say nothing that relates to being alive, which is what the question is asking.

So we add the following facts:

8) Alive means not dead.
   x: t: [alive(x, t) → ¬ dead(x, t)] [¬ dead(x, t) → alive(x, t)]
9) If someone dies, then he is dead at all later times.
   x: t1: At2*: died(x, t1) gt(t2, t1) → dead(x, t2)*

So, Now let's attempt to answer the question "Is Marcus alive?" by proving: ¬ ***alive(Marcus, now)***

Resolution

Propositional Resolution

1. Convert all the propositions of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
    1. Select two clauses. Call these the parent clauses.
    2. Resolve them together. The resulting clause, called the resolvent, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and ¬ *L* such that one of the parent clauses contains *L* and the other contains ¬*L,* then select one such pair and eliminate both *L* and ¬ *L* from the resolvent.
    3. If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of classes available to the procedure.

The Unification Algorithm

- In propositional logic, it is easy to determine that two literals cannot both be true at the same time.
- Simply look for L and ¬L in predicate logic, this 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 the man(John) and ¬man(Spot) 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 recursive procedure, called the unification algorithm, that does it.

Algorithm: Unify(L1, L2)

1. If L1 or L2 are both variables or constants, then:
    1. If L1 and L2 are identical, then return NIL.
    2. Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL}, else return (L2/L1).
    3. Also, 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 LI 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 :
    1. Call Unify with the i^{th} argument of L1 and the i^{th} argument of L2, putting the result in S.
    2. If S contains FAIL then return {FAIL}.
    3. If S is not equal to NIL then:
        2. Apply S to the remainder of both L1 and L2.
        3. SUBST: = APPEND(S, SUBST).
6. Return SUBST.

Resolution in Predicate Logic

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements F and a statement to be proved P:

Algorithm: Resolution

1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in 1.
3. Repeat until a contradiction found, no progress can make, or a predetermined amount of effort has expanded.
    1. Select two clauses. Call these the parent clauses.
    2. Resolve them together. The resolvent will the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals T1 and ¬T2 such that one of the parent clauses contains T2 and the other contains T1 and if T1 and T2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We call T1 and T2 Complementary literals. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should omit from the resolvent.
    3. If the resolvent is an empty clause, then a contradiction has found. Moreover, If it is not, then add it to the set of classes available to the procedure.

Resolution Procedure

- Resolution is a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form.
- Resolution produces proofs by refutation.
- In other words, ***to prove a statement (i.e., to show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e., that it is unsatisfiable).***
- The resolution procedure is a simple iterative process: at each step, two clauses, called the parent clauses, are compared (resolved), resulting in a new clause that has inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose that there are two clauses in the system:

winter **V** ***summer***

¬ ***winter*** **V** ***cold***

- Now we observe that precisely one of winter and ¬ winter will be true at any point.
- If winter is true, then cold must be true to guarantee the truth of the second clause. If ¬ winter is true, then summer must be true to guarantee the truth of the first clause.
- Thus we see that from these two clauses we can deduce ***summer V cold***
- This is the deduction that the resolution procedure will make.
- Resolution operates by taking two clauses that each contains the same literal, in this example, ***winter***.
- Moreover, The literal must occur in the positive form in one clause and in negative form in the other. The resolvent obtained by combining all of the literals of the two parent clauses except the ones that cancel.
- If the clause that produced is the empty clause, then a contradiction has found.

For example, the two clauses

winter

¬ winter

will produce the empty clause.

Natural Deduction Using Rules

Testing whether a proposition is a tautology by testing every possible truth assignment is expensive—there are exponentially many. We need a **deductive system**, which will allow us to construct proofs of tautologies in a step-by-step fashion.

The system we will use is known as **natural deduction**. The system consists of a set of **rules of inference** for deriving consequences from premises. One builds a proof tree whose root is the proposition to be proved and whose leaves are the initial assumptions or axioms (for proof trees, we usually draw the root at the bottom and the leaves at the top).

For example, one rule of our system is known as **modus ponens**. Intuitively, this says that if we know P is true, and we know that P implies Q, then we can conclude Q.

$$\frac{P \qquad P \Rightarrow Q}{Q} \text{ (modus ponens)}$$

The propositions above the line are called **premises**; the proposition below the line is the **conclusion**. Both the premises and the conclusion may contain metavariables (in this case, P and Q) representing arbitrary propositions. When an inference rule is used as part of a proof, the metavariables are replaced in a consistent way with the appropriate kind of object (in this case, propositions).

Most rules come in one of two flavors: **introduction** or **elimination** rules. Introduction rules introduce the use of a logical operator, and elimination rules eliminate it. Modus ponens is an elimination rule for ⇒. On the right-hand side of a rule, we often write the name of the rule. This is helpful when reading proofs. In this case, we have written (modus ponens). We could also have written (⇒-elim) to indicate that this is the elimination rule for ⇒.

Rules for Conjunction

Conjunction (∧) has an introduction rule and two elimination rules:

$$\frac{P \qquad Q}{P \wedge Q} \text{ (∧-intro)} \qquad \frac{P \wedge Q}{P} \text{ (∧-elim-left)} \qquad \frac{P \wedge Q}{Q} \text{ (∧-elim-right)}$$

Rule for T

The simplest introduction rule is the one for T. It is called "unit". Because it has no premises, this rule is an **axiom**: something that can start a proof.

$$\frac{}{T} \text{ (unit)}$$

Rules for Implication

In natural deduction, to prove an implication of the form P ⇒ Q, we assume P, then reason under that assumption to try to derive Q. If we are successful, then we can conclude that P ⇒ Q.

In a proof, we are always allowed to introduce a new assumption P, then reason under that assumption. We must give the assumption a name; we have used the name x in the example below. Each distinct assumption must have a different name.

$$\frac{}{[x : P]} \text{ (assum)}$$

Because it has no premises, this rule can also start a proof. It can be used as if the proposition P were proved. The name of the assumption is also indicated here.
However, you do not get to make assumptions for free! To get a complete proof, all assumptions must be eventually *discharged*. This is done in the implication introduction rule. This rule introduces an implication $P \Rightarrow Q$ by discharging a prior assumption [x : P]. Intuitively, if Q can be proved under the assumption P, then the implication $P \Rightarrow Q$ holds without any assumptions. We write x in the rule name to show which assumption is discharged. This rule and modus ponens are the introduction and elimination rules for implications.

$$\frac{\begin{array}{c}[x : P] \\ \vdots \\ Q\end{array}}{P \Rightarrow Q}\ (\Rightarrow\text{-intro/x}) \qquad \frac{P \qquad P \Rightarrow Q}{Q}\ (\Rightarrow\text{-elim, modus ponens})$$

A proof is valid only if every assumption is eventually discharged. This must happen in the proof tree below the assumption. The same assumption can be used more than once.

Rules for Disjunction

$$\frac{P}{P \vee Q}\ (\vee\text{-intro-left}) \qquad \frac{Q}{P \vee Q}\ (\vee\text{-intro-right}) \qquad \frac{P \vee Q \qquad P \Rightarrow R \qquad Q \Rightarrow R}{R}\ (\vee\text{-elim})$$

Rules for Negation

A negation $\neg P$ can be considered an abbreviation for $P \Rightarrow \bot$:

$$\frac{P \Rightarrow \bot}{\neg P}\ (\neg\text{-intro}) \qquad \frac{\neg P}{P \Rightarrow \bot}\ (\neg\text{-elim})$$

Rules for Falsity

$$\frac{\begin{array}{c}[x : \neg P] \\ \vdots \\ \bot\end{array}}{P}\ (\text{reductio ad absurdum, RAA/x}) \qquad \frac{\bot}{P}\ (\text{ex falso quodlibet, EFQ})$$

Reductio ad absurdum (RAA) is an interesting rule. It embodies proofs by contradiction. It says that if by assuming that P is false we can derive a contradiction, then P must be true. The assumption x is discharged in the application of this rule. This rule is present in classical logic but not in **intuitionistic** (constructive) logic. In intuitionistic logic, a proposition is not considered true simply because its negation is false.

Excluded Middle

Another classical tautology that is not intuitionistically valid is the **the law of the excluded middle**, $P \vee \neg P$. We will take it as an axiom in our system. The Latin name for this rule is *tertium non datur*, but we will call it *magic*.

$$\frac{}{P \vee \neg P}\ (\text{magic})$$

Proofs

A proof of proposition P in natural deduction starts from axioms and assumptions and derives P with all assumptions discharged. Every step in the proof is an instance of an inference rule with metavariables substituted consistently with expressions of the appropriate syntactic class.

Example

For example, here is a proof of the proposition (A ⇒ B ⇒ C) ⇒ (A ∧ B ⇒ C).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[y : A \wedge B]}\,(\text{A})}{A}\,(\wedge\text{E}) \quad \dfrac{}{[x : A \Rightarrow B \Rightarrow C]}\,(\text{A})}{B \Rightarrow C}\,(\Rightarrow\text{E}) \quad \dfrac{}{[y : A \wedge B]}\,(\text{A})}{B}\,(\wedge\text{E})}{C}\,(\Rightarrow\text{E})}{A \wedge B \Rightarrow C}\,(\Rightarrow\text{I,y})}{(A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)}\,(\Rightarrow\text{I,x})$$

The final step in the proof is to derive (A ⇒ B ⇒ C) ⇒ (A ∧ B ⇒ C) from (A ∧ B ⇒ C), which is done using the rule (⇒-intro), discharging the assumption [x : A ⇒ B ⇒ C]. To see how this rule generates the proof step, substitute for the metavariables P, Q, x in the rule as follows: P = (A ⇒ B ⇒ C), Q = (A ∧ B ⇒ C), and x = x. The immediately previous step uses the same rule, but with a different substitution: P = A ∧ B, Q = C, x = y.
The proof tree for this example has the following form, with the proved proposition at the root and axioms and assumptions at the leaves.

A proposition that has a complete proof in a deductive system is called a **theorem** of that system.
Soundness and Completeness
A measure of a deductive system's power is whether it is powerful enough to prove all true statements. A deductive system is said to be **complete** if all true statements are theorems (have proofs in the system). For propositional logic and natural deduction, this means that all tautologies must have natural deduction proofs. Conversely, a deductive system is called **sound** if all theorems are true. The proof rules we have given above are in fact sound and complete for propositional logic: every theorem is a tautology, and every tautology is a theorem. Finding a proof for a given tautology can be difficult. But once the proof is found, checking that it is indeed a proof is completely mechanical, requiring no intelligence or insight whatsoever. It is therefore a very strong argument that the thing proved is in fact true.
We can also make writing proofs less tedious by adding more rules that provide reasoning shortcuts. These rules are sound if there is a way to convert a proof using them into a proof using the original rules. Such added rules are called **admissible**.