SCSX1021 ARTIFICIAL INTELLIGENCE

Unit:III

KNOWLEDGE AND REASONING

Logical Agents

The **representation** of knowledge and the **reasoning** processes that bring knowledge to life are central to the entire field of artificial intelligence.

• Knowledge and reasoning are enable successful behaviors that would be very hard to achieve.

• Knowledge and reasoning also play a crucial role in dealing with **partially observable environments.**

• A knowledge-based agent can combine general knowledge with current percepts to infer hidden aspects of the current state prior to selecting actions.

KNOWLEDGE BASED AGENTS

• The central component of a knowledge-based agent is its knowledge base, or KB.

• A knowledge base is a set of **sentences**.

• **Knowledge Representation Language:-**Each sentence is expressed in a language called a Knowledge Representation Language and represents some assertion about the world.

• **Inferenc**e :- There is a way to add new sentences to the knowledge base is **TELL** and to query what is know is **ASK**.

TELL :- A way to add new sentences to the knowledge base

ASK :- A way to query what is known.

Both tasks may involve **INFERENCE** that is, deriving new sentences from old.

In LOGICAL AGENTS, inference must obey the fundamental requirement that when one ASKS a question of the knowledge base, the answer should follow from what has been told (or rather, TELLED) to the knowledge base previously.

Knowledge Based Agent Program

```
function KB-AGEN T(percept) returns an action
static: KB, a knowledge base
        t, a counter, initially 0, indicating time
        TELL(KB, MAKE-PERCEPT-SENTENCE(percept ,t))
        action ← AsK(KB, MAKE-ACTION- QUERY(t))
        TELL(KB, MAKE-ACTION-SENTENCE(action,t))
        T ←t+1
return action
```

Like all our agents, it takes a percept as input and returns an action.

- The agent maintains a knowledge base, KB, which may initially contain some background knowledge. Each time the agent program is called, it does three things.
  - First, it TELLS the knowledge base what it perceives.
  - Second, it ASKS the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on.
  - Third, the agent records its choice with TELL and executes the action. The TELL is necessary to let the knowledge base know that the hypothetical action has actually been executed.

The details of the Representation Language are hidden inside three functions that implement the interface between the sensors and actuators and the core representation and reasoning system.

- MAKE-PERCEPT-SENTENCE takes a percept and a time and returns a sentence asserting that the agent perceived the given percept at the given time.
- MAKE-ACTION-QUERY takes a time as input and returns a sentence that asks what action should be done at the current time.

• MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed. The details of the inference mechanisms are hidden inside TELL and ASK.

Knowledge level

The knowledge-based agent is not an arbitrary program for calculating actions. It is amenable to a description at the knowledge level, where we need specify only what the agent knows and what its goals are, in order to fix its behavior.

Implementation level

For example : an automated taxi might have the goal of dropping a passenger to Marin County and might know that it is in San Francisco and that the Golden Gate Bridge is the only link between the two locations. Then we can expect it to cross the Golden Gate Bridge because it knows that that will achieve its goal. This analysis is independent of how the taxi works at the implementation level. It doesn't matter whether its geographical knowledge is implemented as linked lists or pixel maps, or whether it reasons by manipulating strings of symbols stored in registers or by propagating noisy signals in a network of neurons.

• One can build a knowledge-based agent simply by Telling it what it needs to know.

• The agent's initial program, before it starts to receive percepts, is built by adding one by one the sentences that represent the designer's knowledge of the environment.

• Designing the representation language to make it easy to express this knowledge in the form of sentences simplifies the construction problem enormously. This is called the **declarative approach** to system building.

• The **procedural approach** encodes desired behaviors directly as program code; minimizing the role of explicit representation and reasoning can result in a much more efficient system

The Wumpus world environment

The **wumpus world** is a cave consisting of rooms connected by passageways.

• Lurking somewhere in the cave is the wumpus, a beast that eats anyone who enters its room.

• The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in).

• The only mitigating feature of living in this environment is the possibility of finding a heap of gold.

• Although the wumpus world is rather tame by modern computer game standards, it makes an excellent test bed environment for intelligent agents.

• Michael Genesereth was the first to suggest this.

A Sample Wumpus world is shown in the figure 3.1.

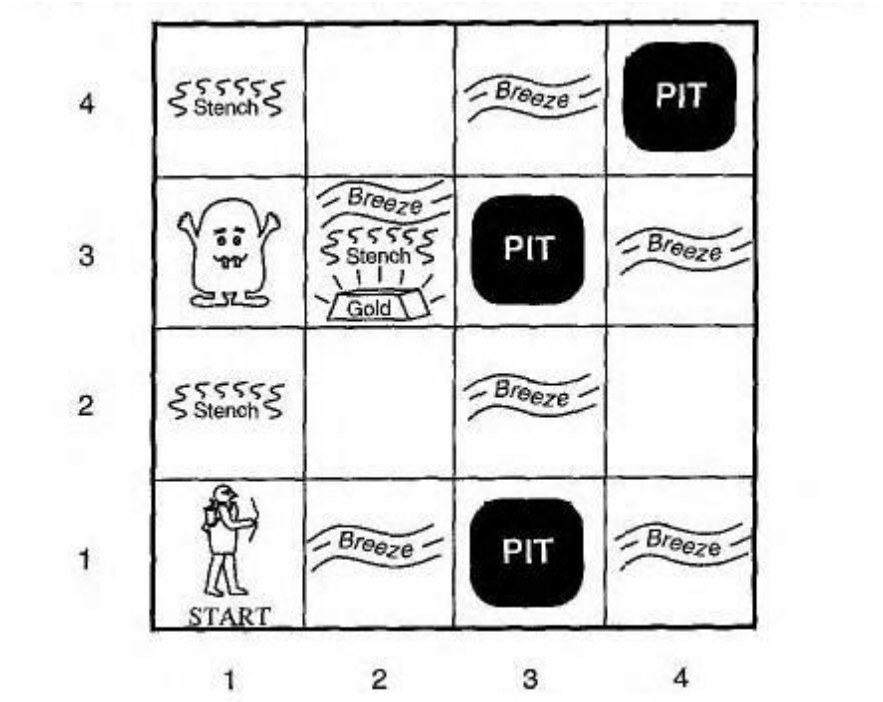


Figure 3.1 Wumpus World

The precise definition of the task environment is given by the PEAS description:

• **Performance measure: +I000 for picking up the gold, -1000 for falling into a pit or** being eaten by the wumpus, -1 for each action taken and -10 for using up the arrow.

• **Environment: A 4 x 4 grid of rooms. The agent always starts in the square labeled** [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.

• **Actuators:** The agent can move forward, turn left by 90°, or turn right by 90°. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. Moving forward has no effect if there is a wall in front of the agent. The action Grab can be used to pick up an object that

is in the same square as the agent. The action Shoot can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent only has one arrow, so only the first Shoot action has any effect.

• **Sensors:** The agent has five sensors, each of which gives a single bit of information.

1. In the square containing the wumpus and in the directly (not diagonally) adjacent squares the agent will perceive a stench.
2. In the squares directly adjacent to a pit, the agent will perceive a breeze.
3. In the square where the gold is, the agent will perceive a glitter.
4. When an agent walks into a wall, it will perceive a bump.
5. When the wumpus is killed, it emits a woeful scream that can be perceived anywhere in the cave.

The percepts will be given to the agent in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent will receive the

percept [Stench, Breeze, None, None, None].

In most instances of the wumpus world, it is possible for the agent to retrieve the gold safely. Occasionally, the agent must choose between going home empty-handed and risking death to find the gold. About 21% of the environments are utterly unfair, because the gold is in a pit or surrounded by pits.

The agent's initial knowledge base contains the rules of the environment, as listed above; in particular, it knows that it is in [l,l] and that [l,l] is a safe square.

• The first percept is [None, None, None, None, None], from which the agent can conclude that its neighboring squares are safe. Figure 3.2(a) shows the agent's state of knowledge at this point. We list the sentences in the kowledge base using letters such as B (breezy) and OK (safe, neither pit nor wumpus) marked in the appropriate squares.

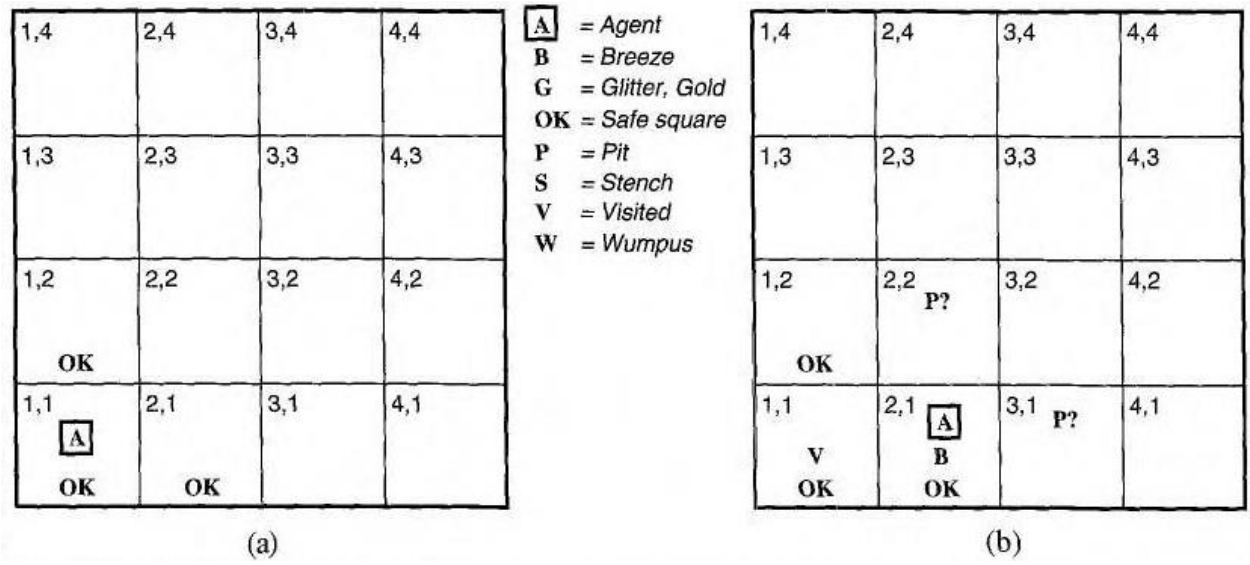


Figure 3.2 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [None, None, None, None, None]. (b) After one move, with percept [None, Breeze, None, None, None].

From the fact that there was no stench or breeze in [1,1], the agent can infer that [1,2] and [2,1] are free of dangers. They are marked with an OK to indicate this. A cautious agent will move only into a square that it knows is OK. Let us, suppose the agent decides to move forward to [2,1], giving the scene in Figure 3.2(b).

• The agent detects a breeze in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1, 1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation P? in Figure 3.2(b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and has not been visited yet. So the prudent agent will turn around, go back to [1, 1] and then proceed to [1,2].

• The new percept in [1,2] is [Stench, None, None, None, None], resulting in the state of knowledge shown in Figure 3.2(a). The stench in [1,2] means that there must be a wumpus nearby. But the wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]).

Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this. Moreover, the lack of a Breeze in [1,2] implies that there is no pit in [2,2]. Yet we already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different

places and relies on the lack of a percept to make one crucial step. The inference is beyond the abilities of most animals, but it is typical of the kind of reasoning that a logical agent does.

• The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. We will not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us Figure 3.3(b). In [2,3], the agent detects a glitter, so it should grab the gold and thereby end the game.

• In each case where the agent draws a conclusion from the available information, that conclusion is guaranteed to be correct the available information is correct. This is a fundamental property of logical reasoning.

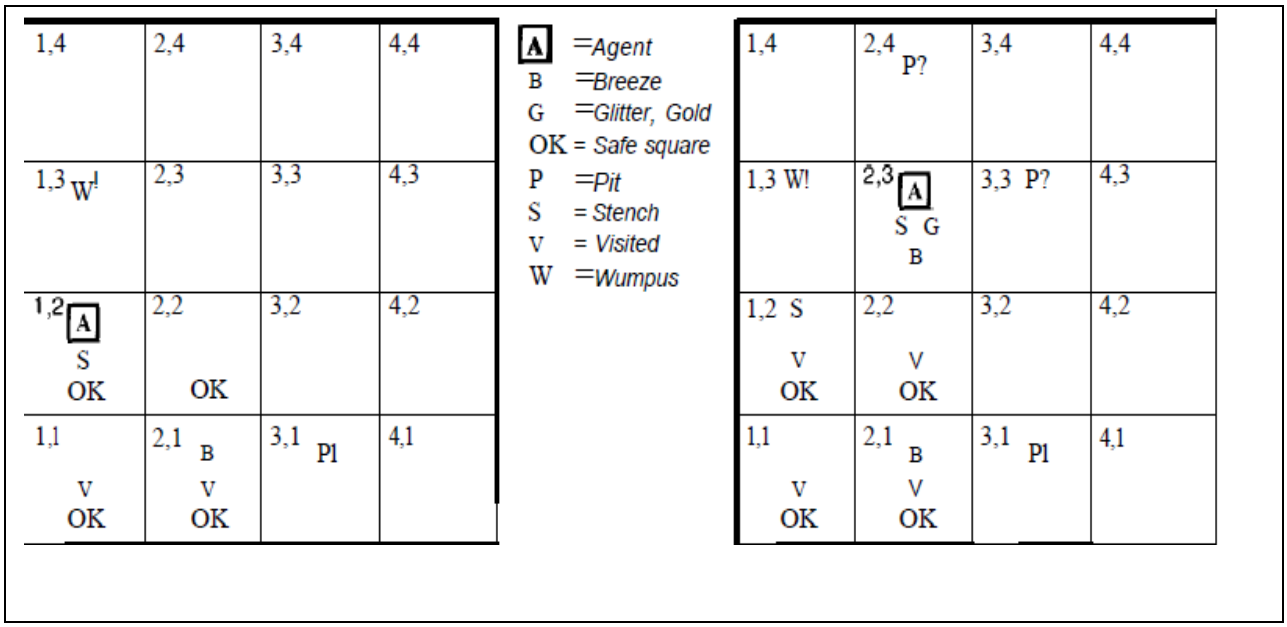


Fig 3.3 Two later stages in the progress of the agent. (a) After the third move, with percept [Stench, None, None, None, None]. (b) After the fifth move, with percept [Stench, Breeze, Glitter, None, None].

LOGIC

• Knowledge bases consists of sentences. These sentences are expressed according to the syntax of the representation language, which specifies all the sentences that are well formed.

• The syntax is clear enough in ordinary arithmetic:

"x + y = 4" is a well-formed sentence, whereas "x2y+ =" is not.

semantics

• A logic must also define the semantics of the language.

• Semantics means "meaning" of sentences. In logic, the definition is more precise.

• The semantics of the language defines the truth of each sentence with respect to each possible world.

• For example, "x + y = 4" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1.1 .

In standard logics, every sentence must be either true or false in each possible world-there is no "in between.

Example : Assume x and y as the number of men and women sitting at a table playing bridge, for example, and the sentence x + y = 4 is true when there are four in total; formally, the possible models are just all possible assignments of numbers to the variables x and y. Each such assignment fixes the truth of any sentence of arithmetic whose variables are x and y.

Entailment

• Entailment means that one thing follows from another. Relation of logical entailment between sentences is involved that a sentence follows logical from another sentence.

• KB ⊨ α Knowledge base KB entails sentence α if and only if α is true in all worlds where KB is true

– E.g., the KB containing “the Giants won” and “the Reds won” entails “Either the Giants won or the Reds won”.

– E.g., x+y = 4 entails 4 = x+y.

– Entailment is a relationship between sentences (i.e., syntax) that is based on semantics.

Models

• Logicians typically think in terms of models, which are formally structured worlds with respect to which truth can be evaluated.

m is a model of a sentence α if α is true in m

M(α) is the set of all models of α

Then KB ⊨ α if M(KB) ⊆ M(α)

– E.g. KB=Giants won and Reds won α = Giants won

Consider the situation in Figure 3.2(b): the agent has detected nothing in [1,1] and a breeze in [2,1]. These percepts, combined with the agent's knowledge of the rules of the wumpus world, constitute the KB. The agent is interested (among other things) in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits. Each of the three squares might or might not contain a pit, so (for the purposes of this example) there are $2^3 = 8$ possible models. These are shown in Figure 3.3.

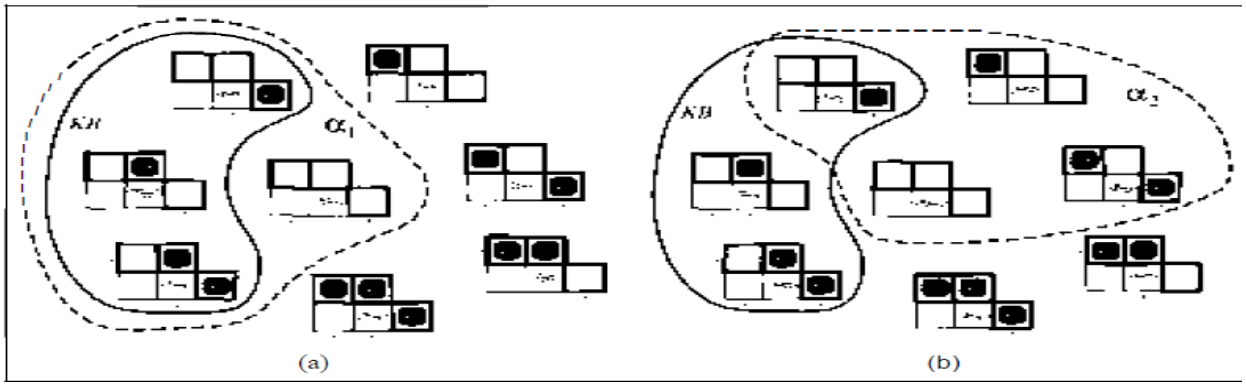


Figure 3.3 : Possible models for the presence of pits in squares [1,2], [2,2], and [3,1], given observations of nothing in [1,1] and a breeze in [2,1]. (a) Models of the knowledge base and αl (no pit in [1,2]). (b) Models of the knowledge base and α2 (no pit in [2,2]).

Conclusion

• The KB is false in models that contradict what the agent knows-for example, the KB is false in any model in which [l,2] contains a pit, because there is no breeze in [1,1].

• Now let us consider two possible conclusions:

• α1= "There is no pit in [1,2].“

• α2= "There is no pit in [2,2].“

• In every model in which KB is true, α1 is also true.

• Hence, KB ⊨ α1 : there is no pit in [1,2].

• In some models in which KB is true, α2 is false.

• Hence, KB ¥ α2 :so, the agent cannot conclude that there is no pit in [2,2].

Logical Inference : The above shown example illustrates entailment and also show how the definition of entailment can be applied to derived conclusions to carry out logical inference.

Model Checking : The inference algorithm illustrated in Figure 3.3 is called model checking, because it enumerates all possible models to check the α is true in all models in which KB is true.

If an inference algorithm i can derive α from KB, then

KB $\vdash_i \alpha$,

Pronunciation for above notation is :

"α is derived from KB by i" (or) " i derives α from KB".

Sound : An inference algorithm that derives only entailed sentences is called sound or truth preserving.

• i is sound if whenever KB $\vdash_i \alpha$, it is also true that KB $\models \alpha$

Soundness is a highly desirable property. An unsound inference procedure essentially makes things up as it goes along-it announces the discovery of nonexistent needles.

Complete: an inference algorithm is complete if it can derive any sentence that is entailed.

• i is complete if whenever KB $\models \alpha$, it is also true that KB $\vdash_i \alpha$

PROPOSITIONAL LOGIC

• Propositional logic is the simplest logic.

• The syntax of propositional logic and its semantics-the way in which the truth of sentences is determined.

Syntax -The syntax of propositional logic defines the allowable sentences.

The atomic sentences - the indivisible syntactic elements consist of a single proposition symbol. Each such symbol stands for a proposition that can be true or false.

Rules:

i. Uppercase names are used for symbols (ie) P,Q,R and so on.

ii. Names are Arbitrary

Complex Sentences : Complex sentences are constructed from simpler sentences using logical connections. There are five connectives.

- If S is a sentence, ¬S is a sentence (negation)
- If S1 and S2 are sentences, S1 ∧S2 is a sentence (conjunction)
- If S1 and S2 are sentences, S1 ∨S2 is a sentence (disjunction)
- If S1 and S2 are sentences, S1 ⇒ S2 is a sentence (implication)
- If S1 and S2 are sentences, S1 ⇔ S2 is a sentence (biconditional / IF AND ONLY IF)

¬ (not). A sentence such as $\neg W_{1,3}$ is called the negation of $W_{1,3}$. A literal is either an atomic sentence (a positive literal) or a negated atomic sentence (a negative literal).

/\ (and). A sentence whose main connective is /\, such as $W_{1,3} /\backslash P_{3,1}$, is called a conjunction; its parts are the conjuncts. (The /\ looks like an "N' for "And.")

V (or). A sentence using V, such as $(W_{1,3}/\backslash P_{3,1}) VW_{2,2}$, is a disjunction of the Disjuncts $(W_{1,3} /\backslash P_{3,1})$ and $W_{2,2}$.

⇒ (implies). A sentence such as $(W_{1,3}/\backslash P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an implication (or conditional). Its premise or antecedent is $(W_{1,3}/\backslash P_{3,1})$,and its conclusion or consequent is $W_{2,2}$. Implications are also known as rules or if-then statements.

⇔ (if and only if). The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a biconditional.

BNF(Backus-Naur Form): The figure 3.4 shows about the BNF grammar of sentence in propositional logic.

Sentence	→	*Atomicbentence* | *ComplexSentence*
Atomicilentence	→	True | False | *Symbol*
Symbol	→	P | Q | R | .. ,
ComplexSentence	→	¬ *Sentence*
		(Sentence 1\ Sentence)
		(Sentence V Sentence)
		(Sentence ⇒ Sentence)
		(Sentence ⇔ Sentence)

Figure 3.4 A BNF (Backus-Naur Form) grammar of sentences in propositional logic.

Every sentence constructed with binary connectives must be enclosed in parentheses. This ensures that we have to write ((A /\ B) ⇒ C) instead of A /\ B ⇒C.

• Order of precedence for the connectives is similar to the precedence used in arithmetic.

For example, ab+ c is read as ((ab) + c) rather than a(b + c) because multiplication has higher precedence than addition.

The order of precedence in propositional logic

• The order of precedence in propositional logic is (from highest to lowest) ¬, /\, V, ⇒, ⇔.

• Hence, the sentence ¬P V Q /\ R⇒ S is equivalent to the sentence ((¬P) V (Q /\ R)) ⇒ S.

• Precedence does not resolve ambiguity in sentences such as A /\ B /\ C, which could be read as ((A /\ B) /\ C) or as (A /\ (B /\ C)). These two readings mean the same thing according to the semantics.

Semantics

• The semantics defines the rules for determining the truth of a sentence with respect to a particular model.

• In propositional logic, a model simply fixes the truth value, true or false for every proposition symbol.

• For example, if the sentences in the knowledge base make use of the proposition symbols $P_{l,2}$, $P_{2,2}$, and $P_{3,1}$, then one possible model is

$m1 = \{P_{1,2} = \text{false}, P_{2,2} = \text{false}, P_{3,1} = \text{true}\}$.

• With three proposition symbols, there are $2^3 = 8$ possible models.

• The semantics for propositional logic must specify how to compute the truth value of any sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives.

Atomic sentences

Atomic sentences are easy:

• True is true in every model and False is false in every model.

• The truth value of every other proposition symbol must be specified directly in the model.

• For example, in the model m1 given earlier, $P_{1,2}$ is false.

Complex sentences

• For any sentence s and any model m, the sentence ¬s is true in m if and only if s is false in m.

• Such rules reduce the truth of a complex sentence to the truth of simpler sentences.

• The rules for each connective can be summarized in a truth table that specifies the truth value of a complex sentence for each possible assignment of truth values to its components.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
$false$	$false$	$true$	$false$	$false$	$true$	$true$
$false$	$true$	$true$	$false$	$true$	$true$	$false$
$true$	$false$	$false$	$false$	$true$	$false$	$false$
$true$	$true$	$false$	$true$	$true$	$true$	$true$

Figure 3.5 Truth tables for the logical connectives

Using the above figure 3.5, the truth value of any sentence S can be computed with respect to any model m by a simple process of recursive evaluation. Each model specifies true/false for each proposition symbol

E.g. $P_{1,2}$ $P_{2,2}$, $P_{3,1}$

false true false

With these symbols, 8 possible models, can be enumerated automatically.

Rules for evaluating truth with respect to a model *m*:

¬S is true if S is false

S1/\ S2 is true if S1 is true and S2 is true

S1 v S2 is true if S1is true or S2 is true

S1 ⇒ S2 is true if S1 is false or S2 is true

i.e., is false if S1 is true and S2 is false

S1 ⇔ S2 is true if S1⇒S2 is true and S2⇒S1 is true

Simple recursive process evaluates an arbitrary sentence, e.g.,

¬P1,2 ∧(P2,2)∨ (P3,1) = true∧(true V false) = true ∨ true = true

Confusion:

(i) *P*V*Q* is true when *P* is true or *Q* is true *or both.* There is a different connective called "exclusive or" ("xor" for short) that yields false when both disjuncts are true. There is no consensus on the symbol for exclusive or; two choices are

(ii) Any implication is true whenever its antecedent is false.

For example,

(a) "5 is even implies Sam is smart" is true, regardless of whether Sam is smart.

(b) *"P ⇒Q"* saying that "If *P* is true, then *Q* is true".

This sentence could be *false* if *P* is true but *Q* is false.

From the truth table, Bidirectional P ⇔ Q, it is true whenever both P ⇒ Q and Q ⇒ P are true (ie) "P if and only if Q" or "P if Q".

Example (Wumpus World)

A square is breezy *if* a neighboring square has a pit, and a square is breezy *only if* a neighboring square has a pit. So we need biconditionals such as

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ that $B_{1,1}$ means that there is a breeze in [1,1], $P_{1,2}$ means Pit in [1,2].

From the truth table, that the one-way implication

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ is true but incomplete. $B_{1,1}$, is false and $P_{1,1}$ is true, which violate the rules of the Wumpus world.

To solve this, implication requires the presence of pits if there is a breeze, whereas the biconditional also requires the absence of pits if there is no breeze.

A Simple knowledge base

We have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world.

Let $P_{i,j}$ be true if there is a pit in [i, j].

Let $B_{i,j}$ be true if there is a breeze in [i, j].

The knowledge base includes the following sentences, each one labeled for convenience:

• There is no pit in [l,l]:

R1: ¬P1,1

• A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:

R2 : B1,1 ⇔ (P1,2 V P2,1)

R3: B2,1 ⇔ (P1,1 VP2,2 V P3,1)

• The preceding sentences are true in all wumpus worlds. Now we include the breeze percepts for the first two squares visited in the specific world the agent is in, leading up to the situation.

R4: ¬B1,1

R5: B2,1

The knowledge base, then, consists of sentences *R*1 through *R5*.

INFERENCE RULES

Logical inference is to decide whether $KB \models a$ a for some sentence *a*. Inference algorithm for will be a direct implementation of the definition of entailment: enumerate the models, and check that a is true in every model in which *KB* is true.

Example : Wumpus World

Propositional symbols for Wumpus are B1,1, *B2,1,* P1,1, *P1,2*, *P2,1, P2,2,* and *P3,1*. With the above symbols are 2^7 = 128 possible models. KB is true in any one of these three which is shown in the below Figure 3.6.

BI,I	$B_{2,1}$	PI,1	$P_{I,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_I	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 3.6 . A truth table constructed for the knowledge base given in the text.

From the above table, *KB* is true if *R1* through *R 5* are true, which occurs in just 3 of the 128 rows. In all 3 rows, *P1 ,2* is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

The algorithm for deciding entailment in propositional logic is shown below.

```
function TT-ENTAILS?(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, symbols, [])
```

```
function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
      ifPL-TRUE?(KB, model) then return PL-TRuE?(a, model)
      else return true
  else do
      P ← FIRST(symbols); rest ← REST(symbols)
      return TT-CHECK-ALL(KB, α, rest, EXTEND(P, true, model) and
             TT-CHECK-ALL(KB, α, rest, EXTEND(P, false, model)
```

Equivalence, validity, and satisfiability

Logical equivalence: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as a $\alpha \Leftrightarrow \beta$.

Example : Two sentences α and β are logically equivalent if they are true in same models:

• $\alpha \equiv ß$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$.

• Equivalence for any two sentences α and β is $\alpha \equiv ß$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$

$$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge$$
$$(\alpha \vee \beta) \equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee$$
$$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge$$
$$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee$$
$$\neg(\neg\alpha) \equiv \alpha \quad \text{double-negation elimination}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta) \quad \text{implication elimination}$$
$$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{de Morgan}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{de Morgan}$$
$$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee$$
$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge$$

The symbols *α,* β and γ stands for arbitrary sentences of propositional logic.

Validity

A sentence is valid if it is true in all models.

e.g., A V¬ A, A ⇒ A, (A ∧ (A ⇒ B)) ⇒ B. These are valid / true statements.

Valid sentences are also known as tautologies-they are *necessarily* true. Because the sentence *True* is true in all models, every valid sentence is logically equivalent to *True.*

For any sentences KB and α , *KB* ⊨ α if and only if (*KB* ⇒ α) is valid. Every valid implication sentence describes an inference.

Satisfiability

A sentence is satisfiable if it is true in *some* model.

For example, the knowledge base given earlier, *(R1 ∧ R2 ∧ R3 ∧ R4 ∧ R s),* is satisfiable because there are three models in which it is true.

If a sentence a is true in a model m, then we say that m satisfies α, or that m is a model of α. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence.

FIRST-ORDER LOGIC

FOL, a representation language of knowledge which is powerful than propositional logic (ie)Boolean logic. FOL is an expressive.

- It follows Procedural approach
- It derive facts from other facts(ie.) dependent

First-order logic (like natural language) assumes the world contains

- Noun- refers to Objects. Eg.Name, place, thing.
- Verb refers to relations
- Some relations are functions
- Relation has only one value

Functions have many values assigned to it.

- Objects: people, houses, numbers, colors, baseball games, wars,
- Properties: red, round, prime, Small
- Relations: bigger than, part of, comes between, …
- Functions: father of, best friend, one more than, plus, …

Example :

1. “Evil King John ruled England in 1200”

   Objects: John, England, 1200; Relation: ruled; Properties: evil, king.

2. "One plus two equals three"

   Objects: one, two, three, one plus two; Relation: equals;

   Function: plus. ("One plus two" is a name for the object that is obtained by applying the function "plus" to the objects "one" and "two." Three is another name for this object.)

3. "Squares neighboring the wumpus are smelly."

   Objects: wumpus, squares; Property: smelly; Relation: neighboring.

Ontological Commitment

• The primary difference between propositional and first-order logic lies in the **ontological commitment** made by each language-that is, what it assumes about the **nature of reality**. Special Purpose logic makes faster Ontological commitment.

Temporal Logic: It assumes that facts hold at particular times and that those times (intervals) are ordered (arranged).

High order logic: It is more expressive than FOL. It allows one to make assertions about all relations.

Epistemological Commitments

A logic that allows the possible states of knowledge that it allows with respect to each fact. In first order logic, a sentence represents a fact and the agent believes the sentence to be true, believes it to be false, or has no opinion. These logics therefore have three possible states of knowledge regarding any sentence. Systems using probability theory, can have any *degree of belief*, ranging from 0 (total disbelief) to 1 (total belief).

Ontological Commitment and Epistemological Commitment of different logics:

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

SYNTAX AND SEMANTICS

Syntax → Procedures

Semantics →Meanings

a) Models for FOL

- Models for propositional logic are just sets of truth values for the proposition symbols (ie 0's and 1's).
- Models for first-order logic are represented in terms of objects and predicates on objects (ie) properties of objects (or) relation between objects.
- The **domain** of a model is the set of objects it contains. These objects are sometimes called **domain elements**.
- **Relation** → related set of tuples of objects
- **Tuple** → is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.

Example: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.

• { (Richard the Lionheart, King John), (King John, Richard the Lionheart) }

From the above example, the underlined words are the objects.

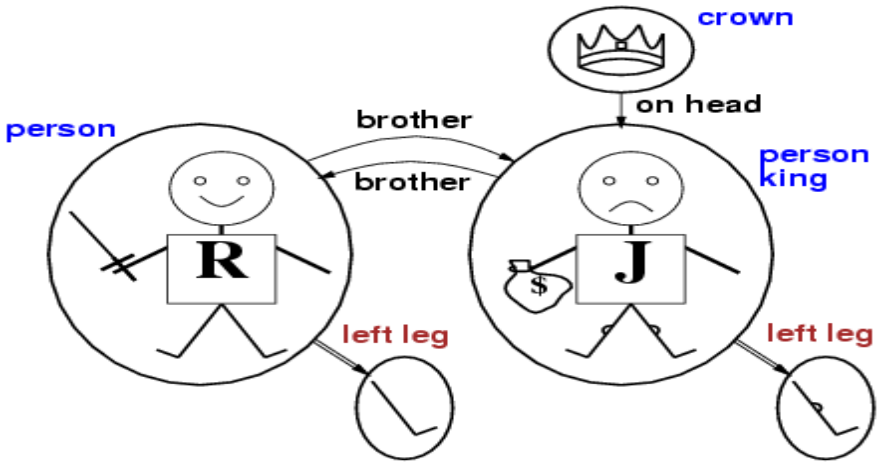


Figure 3.7 A model containing five objects

The above figure 3.7 contains

1) Five objects – Richard, John, Left, Legs, Crown

2) Two binary relations – "Brothers", "on head"

3) Three unary relations (indicated by labels on the objects), - **"Person"** is Richard & John, **"King"** is John and **Crown**

4) one unary function- left-leg - < Richard the LionHeart> → Richard's left leg

< King John> → John's Left leg

<Richard, John> → It is tuple which consists of set of objects in arranged order with angle brackets.

Total functions: Models In FOL requires total function (ie) every input tuple must have a value.

Symbols and interpretations

The basic syntactic elements of first-order logic are the symbols that stand for **objects, relations, and functions.**

The symbols, are in three kinds:

- Constant Symbols, which stand for objects; (*Richard, John)*
- Predicate symbols, which stand for relations; (*Brother, OnHead, Person, King, Crown)*
- Function symbols, which stand for functions. (*LeftLeg)*

- The semantics must relate sentences to models in order to determine truth. For this to happen, we need an interpretation.
- **Interpretation** that specifies exactly which objects, relations, and functions are referred to by the constant, predicate, and function symbols.

Intended Interpretation

Example:

a) **Object** :- Richard refers to Richard the Lionheart and John refers to the evil King John.

b) **Relation** :- Brother refers to the brotherhood relation between Richard and John, that is, the set of tuples of objects given in the above model;

OnHead refers to the "on head" relation that holds between the crown and King John; Person, King, and Crown refer to the sets of objects that are persons, kings, and crowns.

c) **Function** :- LeftLeg refers to the "left leg" function, that is, the mapping given in above model.

For example, one interpretation maps Richard to the crown and John to King John's left leg. There are five objects in the model, so there are 25 possible interpretations just for the constant symbols Richard and John.

Syntax of first-order logic with equality, specified in Backus-Naur form(BNF)

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \\
&\quad \textit{(Sentence Connective Sentence)} \\
&\quad \textit{Quantifier Variable, ... Sentence} \\
&\quad \neg\ \textit{Sentence} \\
\textit{AtomicSentence} &\rightarrow \textit{Predicate(Term, ...)} \mid \textit{Term} = \textit{Term} \\
\textit{Term} &\rightarrow \textit{Function(Term, .. .)} \\
&\quad \textit{Constant} \\
&\quad \textit{Variable} \\
\textit{Connective} &\rightarrow \Rightarrow \mid \wedge \mid \vee \mid \Leftrightarrow \\
\textit{Quantifier} &\rightarrow \forall \mid \exists \\
\textit{Constant} &\rightarrow A \mid \text{XII} \mid \textit{John} \mid \ldots \\
\textit{Variable} &\rightarrow a \mid x \mid s \mid \cdots \\
\textit{Predicate} &\rightarrow \textit{Before} \mid \textit{HasColor} \mid \textit{Raining} \mid \\
\textit{Function} &\rightarrow \textit{Mother} \mid \textit{LeftLeg} \mid \text{'''}
\end{aligned}
$$

Detailed Explanation for all the above elements

1. Terms:

• A term is a logical expression that refers to an object

• To build a term, we use constant symbols, variables and function symbols.

• To build a sentence, Quantifiers and Predicate symbols are used.

• Example: Constant symbol-"King John's left leg" rather than giving a name to his leg.

Function symbol - *LeftLeg(John)*

• Term →f(t1, t2,…tn)

• The function symbol refers to some function in the model. The argument terms t1, t2,…tn refers to objects in the domain d1, d2,…dn.

• Eg. i) John refers to King John.

• ii) Left Leg(John) refers to King John's left leg

2. Atomic Sentences

• Constant symbols and function symbols refers to objects and predicate symbols which further refers to relation is called Atomic Sentence which state facts.

• An Atomic Sentence is formed from a predicate symbol followed by parenthesized list of terms.

Atomic sentence = *predicate* (*term1,...,term n*) or *term1 = term2*

Term = *function* (*term1,...,term n*) or *constant* or *variable*

• Eg. 1) William is the brother of Richard

Brother (William, Richard)

2) Richard's father is married to William's mother

Married (Father(Richard), Mother(William))

• An atomic sentence is true in a given model, under a given interpretation, if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.

3) Complex Sentences

Logical Connectives are used to construct more complex sentences, in which meaning of given sentences has to be satisfied.

• Complex sentences are made from atomic sentences using connectives.

$\neg S$, $S1 \wedge S2$, $S1 \vee S2$, $S1 \Leftrightarrow S2$, $S1 \Rightarrow S2$,

E.g. 1)*Sibling(King John, Richard)* $\Rightarrow$ *Sibling(Richard, King John)*

2) Either Richard is king or John is King.

King(Richard) V King (John)

3) Richard is not king, so it implies John is king

$\neg$King(Richard) $\Rightarrow$ King(John)

4) Quantifiers

• To express properties of entire collections of objects, instead of enumerating the objects by name.

• First-order logic contains two standard quantifiers, called ***Universal*** **(∀) and E*xistential* (∃).**

Universal quantification(∀)

- This quantifier is usually denoted as ∀ and Pronounced as "For All".
- Logical Expression is true for all objects x(variable) in the universe.
- Eg. "All kings are persons" is written in first-order logic as
- ∀x *King (x) Person(x). (ie)* "For all *x*, if *x* is a king, then *x* is a person". X → Variable.
- The symbol x is called **variable**. Eg. LeftLeg(x)
- Ground Term- A term with no variable is called a ground term.
- The sentence ∀*x P*, where *P* is any logical expression, says that *P* is true for every object *x*.

Extended Interpretations

• ∀*x P* is true in a given model under a given interpretation if *P* is true in all possible extended interpretations constructed from the given interpretation, where each extended interpretation specifies a domain element to which *x* refers.

• We can extend the interpretation in five ways:

• *x* → Richard the Lionheart,

• *x* → King John,

• *X* → Richard's left leg,

• $x \rightarrow$ John's left leg,

• $x \rightarrow$ the crown.

• The universally quantified sentence

$\forall x$ *King (x)* $\Rightarrow$ *Person(x)* is true under the original interpretation if the sentence

King (x) $\Rightarrow$ *Person(x)* is true in each of the five extended interpretations.

• The universally quantified sentence is equivalent to asserting the following five sentences:

• Richard the Lionheart is a king $\Rightarrow$ Richard the Lionheart is a person. - True

• King John is a king $\Rightarrow$ King John is a person. - True

• Richard's left leg is a king $\Rightarrow$ Richard's left leg is a person. False

• John's left leg is a king $\Rightarrow$ John's left leg is a person. - False

• The crown is a king $\Rightarrow$ the crown is a person. - False

4.2 Existential quantification($\exists$)

• It makes a statement about **some** object in the universe without naming it, by using an existential quantifier.

• Pronounced as "There Exists" logical expression which is true for some objects x (variable) in the universe.

• Eg.1 "Leesa has a brother who is a dog" which is expressed as

$\exists$x brother(x, Lessa) $\rightarrow$ Dog(x) (ie) Leesa's brother is a dog $\Rightarrow$ Leesa is also a dog and hence x may be replaced by brother of Lessa if it exists.

Eg2: a) Z is a Dog

b) Z is a brother of N

a. Dog (Z)

b. Brother (Z, N)

Eg3. $\exists$ *X Crown(x)* $\Rightarrow$ *OnHead(x, John)*

• Applying the semantics, we see that the sentence says that at least one of the following assertions is true:

• Richard the Lionheart is a crown $\Rightarrow$ Richard the Lionheart is on John's head;

• King John is a crown $\Rightarrow$King John is on John's head;

• Richard's left leg is a crown ⇒ Richard's left leg is on John's head;

• Now an implication is true if both premise and conclusion are true, or if its premise is false**.**

So, in the first assertion, if Richard the LionHeart is not a crown, then the first assertion is true and the existential is satisfied.

Nested Quantifiers

• The sentences are represented using both quantifiers (ie) universal and existential are called Nested Quantifiers. There are two types of sentences in which different nested quantifiers are used.

• Simple sentences use same type of quantifiers.

• Eg: a) Same type : "Brothers are siblings" Note : x,y are brothers

∀x ∀y *Brother(x,y) ⇒Sibling(x,y).*

implies

∀x,y sibiling(x,y) ⇔ sibling(y,x)

equivalence

Example:

"Everybody loves somebody"

∀x∃y Loves(x,y)

"There is someone who is loved by everyone", → ∃y ∀x Loves(x,y)

"Everyone will be loved by some body", → ∀*x (*∃*y Loves(x, y))*

*"*Some one will be loved by everybody" → ∃x(∀y Loves(x,y))

Connections between ∀ and ∃

- The two quantifiers are actually intimately connected with each other, through negation.
- Eg: "Everyone likes ice cream"means that there is no one who does not like ice cream:

∀x *Likes(x,IceCream)* ⇔ ∃⌐x ⌐Likes (x, IceCream).

All X Nobody Dislikes

Because ∀ is a conjunction and ∃ is a disjunction.

De Morgan's rules

∀ is a conjunction and ∃ is a disjunction and its not quantified

Quantified Sentences	Unquantified Sentences
$\forall x \neg P = \neg \exists x P$	$\neg (P \wedge Q) = \neg P \vee \neg Q$
$\neg \forall x P = \exists x \neg P$	$\neg P \wedge \neg Q = \neg (P \vee Q)$
$\forall x P = \neg \exists x \neg P$	$(P \wedge Q) = \neg (\neg P \vee \neg Q)$
$\exists x P = \neg \forall x \neg P$	$(P \vee Q) = \neg (\neg P \wedge \neg Q)$

Assertions and queries in first-order logic

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

TELL(KB, King(John)) . Kb – Knowledge Base

TELL(KB, $\forall$x King(x) $\Rightarrow$Person(x)) .

We can ask questions of the knowledge base using ASK.

For example, ASK(KB, King(John))

Questions asked using ASK are called **queries or goals**

(a) ASK(KB, Person(John)) (ie) to find whether John is a person. It is true.

(b) ASK(KB, $\exists$ x Person(x)). It may be True (or) false.

(ie) ASK KB, that there may be some x who is a person and we solve it by providing such an x, that is called substitution list (or) Binding list.

If there is more than one answer, a list of substitutions can be returned.

The kinship domain

• The domain of family relationships is called kinship domain.

• Eg. "E is the mother of C" and "C is the father of W" and rules such as "One's grandmother is the mother of one's parent.“ (ie) “W’s grandmother is the mother of W’s parent”.

• Kinship domain consists of

(a) Objects – people

(b) Unary predicate – Male and Female

(c) Binary predicate – Parent, Brother, Sister

(d) Function – Father. Mother (Every person has exactly one of these)

(e) Relation – Brotherhood, Sisterhood

For example :

• one's mother is one's female parent:

$\forall m,c$ Mother(c)=m $\Leftrightarrow$ Female(m) Λ Parent(m, c). (ie) m is a mother of c $\Leftrightarrow$m is a Female AND m is a parent of C

• One's husband is one's male spouse:

$\forall$ w, h Husband(h, w) $\Leftrightarrow$ Male(h) Λ Spouse(h, w).

(ie) h is a husband (h,w) $\Leftrightarrow$h is male AND h is a spouse of W

• Male and female are disjoint categories:

$\forall x$ Male(x) $\Leftrightarrow \neg$ Female(x). (ie) x is male $\Leftrightarrow$ x is not a Female

• Parent and child are inverse relations:

$\forall$ p, c Parent (p, c) $\Leftrightarrow$ Child (c, p) .(ie) p is parent of c $\Leftrightarrow$c is child of p

Numbers, sets, and lists

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. 0 is natural numbers or nonnegative integers. We need a predicate NatNum that will be true of natural numbers; we need one constant symbol, 0 and we need one function symbol, S (successor).

PEANOAXIOMS : The Peano axioms define natural numbers and addition. Natural numbers are defined recursively: NatNum(O) .

$\forall n$ NatNum(n) $\Rightarrow$ NatNum(S(n)) .

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

We also need **axioms to constrain the successor function**:

n O $\neq$ S(n). (ie) Zero will not be equal to Natural number.

$\forall m,n$ 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) Λ NatNum(n) $\Rightarrow$ + (S(m),n) = S(+(m,n)).

Generally, adding 0 to any natural number m gives m itself but when we use the binary function symbol "+" in the term + (m, 0). In ordinary mathematics, the term would be written m

+ 0 using infix notation. To make our sentences about numbers easier to read, we will allow the use of infix notation.

SYNTACTIC SUGAR : An extension to or abbreviation of the standard syntax that does not change the semantics.

SETS :

• The domain of mathematical set representation, which consists of

(a) Constant – Empty Set { }

(b) Predicates – Member and subset

(c) Functions – Intersection, Union, and Adjoin

{(s1 ∩ s2) , (s1 ∪ s2), x|s2}

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:

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

2. The empty set has no elements adjoined into it, in other words, there is no way to decompose *EmptySet* 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.

$$\forall x,s\ x \in s \Leftrightarrow [\exists y,s2\}\ (s = \{y|s2\} \wedge (x = y \vee x \in s2))]$$

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

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

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

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

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

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

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

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

Lists

Lists are similar to sets. The element in the list can appear multiple times. The lists are ordered. Constant list is *NIL,* which has no element. *Cons, Append, first and rest* are the functions. *Find* is the predicate.

The empty list is []. The term *Cons(x, y),* where *y* is a nonempty list, is written *[xly]*.

The term *Cons(x, Nil),*is written as *[x]*.

A list of several elements, such as *[A, B,* CJ, corresponds to the nested term *Cons(A, Cons(B, Cons(C, Nil))).*

SITUATION CALCULUS

- The Situation Calculus is a logic formalism designed for representing and reasoning about **dynamical domains**.
    - McCarthy, Hayes 1969
    - Reiter 1991
- In First-Order Logic, sentences are either true or false and stay that way. Nothing is corresponding to any sort of change.
- SitCalc represents changing scenarios as a set of SOL formulae.
- A domain is encoded in SOL by three kind of formulae
    - **Action precondition axioms** and **action effects axioms**
    - **Successor state axioms**, one for each fluent
    - **The foundational axioms** of the situation calculus
- **Situation Calculus : An Example**
- **World**:
    - robot
    - items
    - locations (x,y)
    - moves around the world
    - picks up or drops items
    - some items are too heavy for the robot to pick up
    - some items are fragile so that they break when they are dropped
    - robot can repair any broken item that it is holding
- **Actions**

- **move(x, y):** robot is moving to a new location **(x, y)**
- **pickup(o):** robot picks up an object **o**
- **drop(o):** robot drops the object **o** that holds

- **Situations**
  - Initial situation $\mathbf{S_0}$: no actions have yet occurred
  - A new situation, resulting from the performance of an action a in current situation **s**, is denoted using the function symbol **do(a, s)**.
    - **do(move(*2*, *3*), $\mathbf{S_0}$):** denotes the new situation after the performance of action **move(*2*, *3*)** in initial situation $\mathbf{S_0}$.
    - **do(pickup(*Ball*), do(move(*2*, *3*), $\mathbf{S_0}$))**
    - **do(a,s)** is equal to **do(a',s')** **s=s' and a=a'**
- **Fluents: "properties of the world"**
  - **Relational fluents**
    - Statements whose truth value may change
    - They take a situation as a final argument
    - **is_carrying(o, s)**: robot is carrying object **o** in situation **s**
  - **E.g. Suppose that the robot initially carries nothing**
    - **is_carrying(*Ball*, $\mathbf{S_0}$) : FALSE**
    - **is_carrying(*Ball*, do(pickup(*Ball*), $\mathbf{S_0}$)) : TRUE**
- **Action Preconditions Axioms**
  - Some actions may not be executable in a given situation
  - **Poss(a,s):** special binary predicate
    - denotes the executability of action **a** in situation **s**
  - **Examples:**
  - **Poss(drop(o),s) ↔ is_carrying(o,s)**
  - **Poss(pickup(o),s) ↔ (∀z ¬is_carrying(z,s) ∧¬heavy(o))**
- **Action Effects Axioms**
  - Specify the effects of an action **on the fluents**
  - **Examples:**
  - **Poss(pickup(o),s) → is_carrying(o,do(pickup(o),s))**
  - **Poss(drop(o),s) ∧ fragile(o) → broken(o,do(drop(o),s))**

- Is that enough? No, because of the **frame problem**

BULIDING A KNOWLEDGE BASE

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

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

2. **Assemble the relevant knowledge** – The knowledge engineer might already be an expert in the domain to extract what they know this process is called Knowledge Acquisition. At this stage, the knowledge is not represented and the main idea is is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.

3. **Decide on a vocabulary of predicates, functions, and constants** - Translate the important domain-level concepts into logic-level names. Once the choices have been made, the result is a vocabulary that is known as the ontology of the domain. The word *ontology* means a particular theory of the nature of being or existence. The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.

4. **Encode general knowledge about the domain** - The knowledge engineer writes down the axioms for all the vocabulary terms (ie) enabling the expert to check the content. This step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.

5. **Encode a description of the specific problem instance** - It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology

6. **Pose queries to the inference procedure and get answers** - we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing.

7. **Debug the knowledge base** – Answers will be correct for the knowledge base as written, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting. A debugging process could confirm missing axioms or axioms that are too weak can be identified easily by noticing places where the chain of reasoning stops unexpectedly. To understand this seven-step process better, we now apply it to an extended example-the

domain of electronic circuits.

The Electronic Circuits Domain

We will develop an ontology and knowledge base that allow us to reason about digital circuits of the kind shown in Figure 3.8.

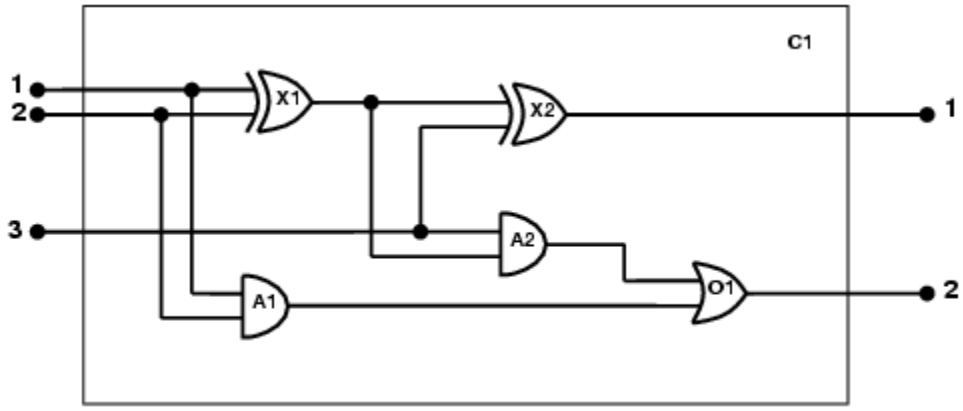


Figure 3.8 A digital circuit C1, purporting to be a one-bit full adder

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

We follow the seven-step process for knowledge engineering

1. Identify the task

   – Does the circuit actually add properly? (circuit verification)

2. Assemble the relevant knowledge

   – Composed of wires and gates; Types of gates (AND, OR, XOR, NOT)

   – Irrelevant: size, shape, color, cost of gates

3. Decide on a vocabulary

   Type(X1) = XOR

   Type(X1, XOR)

   XOR(X1)

4. Encode general knowledge of the domain

   – $\forall t1,t2$ Connected(t1, t2) $\Rightarrow$ Signal(t1) = Signal(t2)

   – $\forall t$ Signal(t) = 1 V Signal(t) = 0

   – $1 \neq 0$

   – $\forall t1,t2$ Connected(t1, t2) $\Leftrightarrow$ Connected(t2, t1)

– ∀g Type(g) = OR ⇒ Signal(Out(1,g)) = 1 ⇔ ∃n Signal(In(n,g)) = 1

– ∀g Type(g) = AND ⇒ Signal(Out(1,g)) = 0 ⇔ ∃n Signal(In(n,g)) = 0

– ∀g Type(g) = XOR ⇒ Signal(Out(1,g)) = 1 ⇔ Signal(In(1,g)) ≠ Signal(In(2,g))

– ∀g Type(g) = NOT ⇒ Signal(Out(1,g)) ≠ Signal(In(1,g))

5. Encode the specific problem instance

Type(X1) = XOR Type(X2) = XOR

Type(A1) = AND Type(A2) = AND

Type(O1) = OR

Connected(Out(1,X1),In(1,X2)) Connected(In(1,C1),In(1,X1))

Connected(Out(1,X1),In(2,A2)) Connected(In(1,C1),In(1,A1))

Connected(Out(1,A2),In(1,O1)) Connected(In(2,C1),In(2,X1))

Connected(Out(1,A1),In(2,O1)) Connected(In(2,C1),In(2,A1))

Connected(Out(1,X2),Out(1,C1)) Connected(In(3,C1),In(2,X2))

Connected(Out(1,O1),Out(2,C1)) Connected(In(3,C1),In(1,A2))

6. Pose queries to the inference procedure

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

∃i1,i2,i3,o1,o2 Signal(In(1,C1)) = i1 Λ Signal(In(2,C1)) = i2 Λ Signal(In(3,C1)) = i3 Λ Signal(Out(1,C1)) = 0 Λ Signal(Out(2,C1)) = 1.

7. Debug the knowledge base May have omitted assertions like 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.

ONTOLOGY

Ontology means Remaining. Representing the abstract concepts is sometimes called **ontological engineering**-it is related to the knowledge engineering process.

In "toy" domains, the choice of representation is not that important; it is easy to come up with a consistent vocabulary.

Real world problems such as shopping on the Internet or controlling a robot in a changing physical environment require more general and flexible representations, we have many choice of representation like Actions, Time, Physical objects and Beliefs so this occur in different domains.

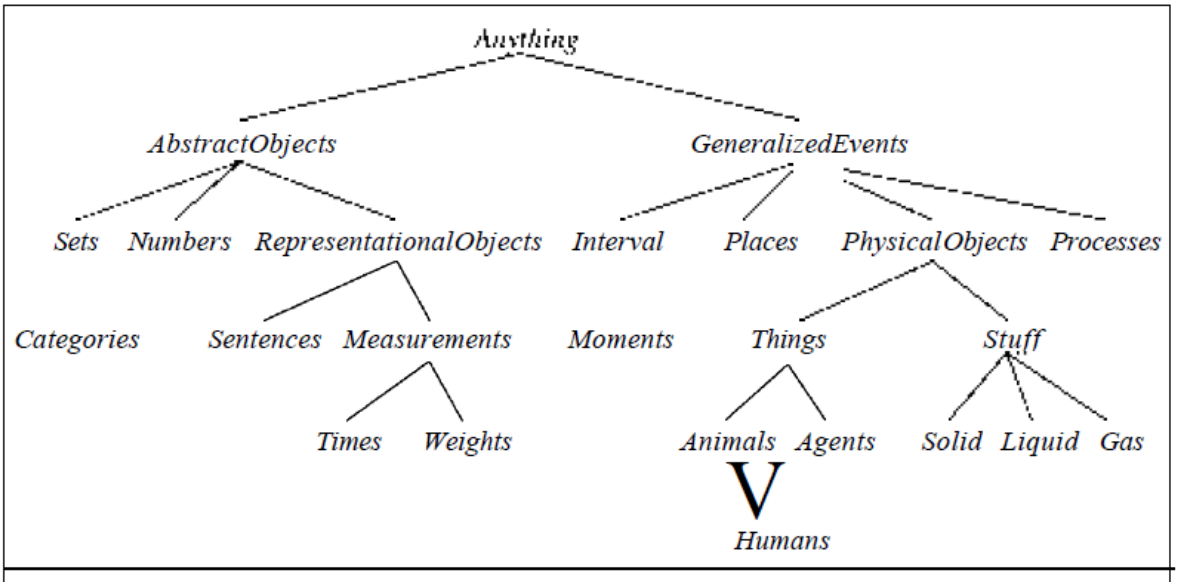


Figure 3.9 The upper ontology of the world

The general framework of concepts is called an **upper ontology,** because of the convention of drawing graphs with the general concepts at the top and the more specific concepts below them, as in above Figure 3.9.

Characteristics of General purpose ontology.

1. A general purpose ontology should be applicable in more or less special purpose domain.
2. In any demanding domain, different areas of knowledge must be unified.

FORWARD AND BACKWARD CHAINING

Forward chaining is one of the two main methods of reasoning when using an inference engine and can be described logically as repeated application of *modus ponens*. Forward chaining is a popular implementation strategy for expert systems, business and production rule systems. The opposite of forward chaining is backward chaining.

Forward chaining starts with the available data and uses inference rules to extract more data (from an end user, for example) until a goal is reached. An inference engine using forward chaining searches the inference rules until it finds one where the antecedent (**If** clause) is known to be true. When such a rule is found, the engine can conclude, or infer, the consequent (**Then** clause), resulting in the addition of new information to its data.

Inference engines will iterate through this process until a goal is reached.

Example:1, suppose that the goal is to conclude the color of a pet named Fritz, given that he croaks and eats flies, and that the rule base contains the following four rules:

1. **If** X croaks and X eats flies - **Then** X is a frog
2. **If** X chirps and X sings - **Then** X is a canary
3. **If** X is a frog - **Then** X is green
4. **If** X is a canary - **Then** X is yellow

Let us illustrate forward chaining by following the pattern of a computer as it evaluates the rules. Assume the following facts:

- Fritz croaks
- Fritz eats flies

With forward reasoning, the inference engine can derive that Fritz is green in a series of steps:

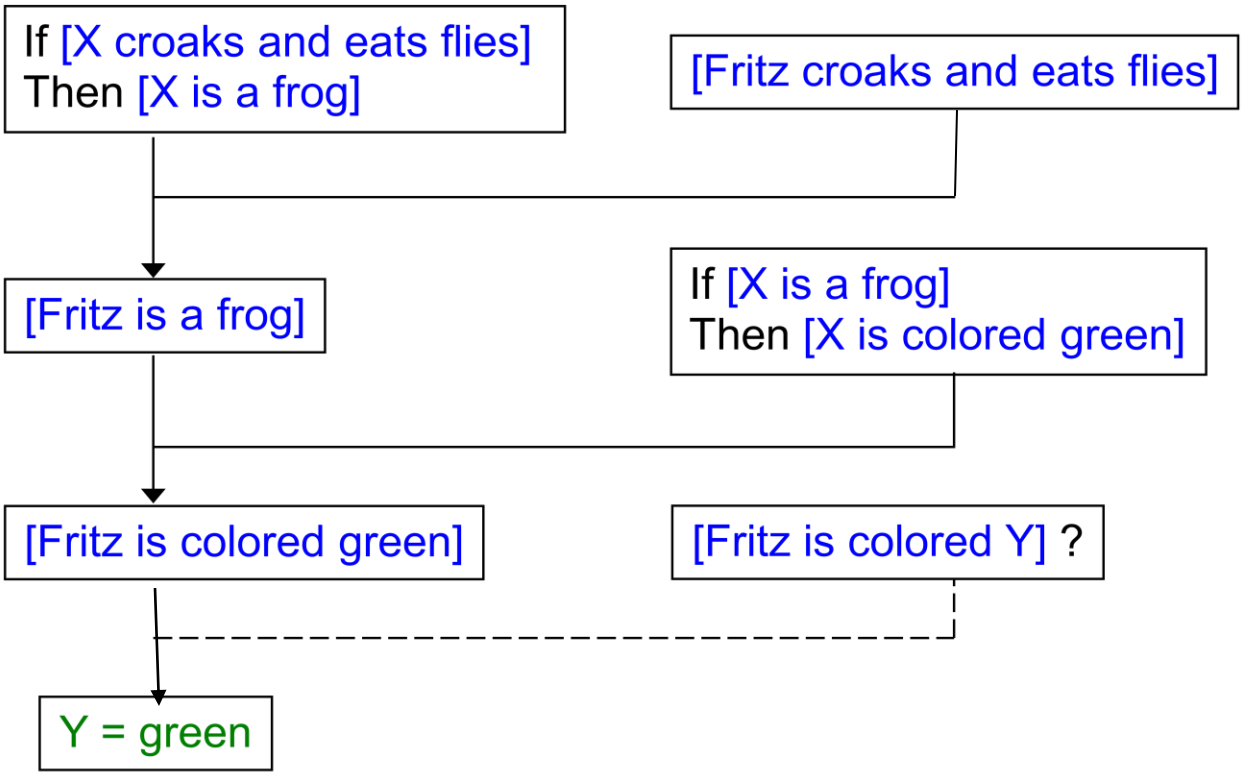
1. Since the base facts indicate that "Fritz croaks" and "Fritz eats flies", the antecedent of rule #1 is satisfied by substituting Fritz for X, and the inference engine concludes:

Fritz is a frog

2. The antecedent of rule #3 is then satisfied by substituting Fritz for X, and the inference engine concludes:

Fritz is green

The name "forward chaining" comes from the fact that the inference engine starts with the data and reasons its way to the answer, as opposed to backward chaining, which works the other way around. In the derivation, the rules are used in the opposite order as compared to backward chaining. In this example, rules #2 and #4 were not used in determining that Fritz is green.



Example:2

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.

To prove:

"West is a criminal".

Proof :

First, represent these facts as first-order definite clauses. The forward-chaining algorithm solves this problem.

Step 1: " it is a crime for an American to sell weapons to hostile nations"

American(x) $\wedge$ Weapon(y) $\wedge$ Sells(x,y,z) $\wedge$ Hostile(z) $\Rightarrow$ Criminal(x)

Step 2: " Nono … has some missiles", Missile(x): is transformed in to 2 definite clauses :

$\wedge$x Owns(Nono,x) $\exists$i.e., Owns(Nono,M1) and Missile(M1)

where M1 is a new constant

Step 3: "… all of its missiles were sold to it by Colonel West"

Sells(West, x, Nono)$\Rightarrow$ Owns(Nono ,x) $\wedge$Missile(x)

Step 4: Missiles are weapons:

$$Weapon(x) \Rightarrow Missile(x)$$

Step 5:An enemy of America counts as "hostile“:

$$Hostile(x) \Rightarrow Enemy(x,America)$$

Step 6: West, who is American …

$$American(West)$$

Step 7: The country Nono, an enemy of America …

$$Enemy(Nono, America)$$

This knowledge base contains no function symbols and is therefore an instance of the class of Datalog knowledge bases-that is, sets of first-order definite clauses with no function symbols. The diagrammatic representation of Forward Chaining shown in the below figure 3.10.

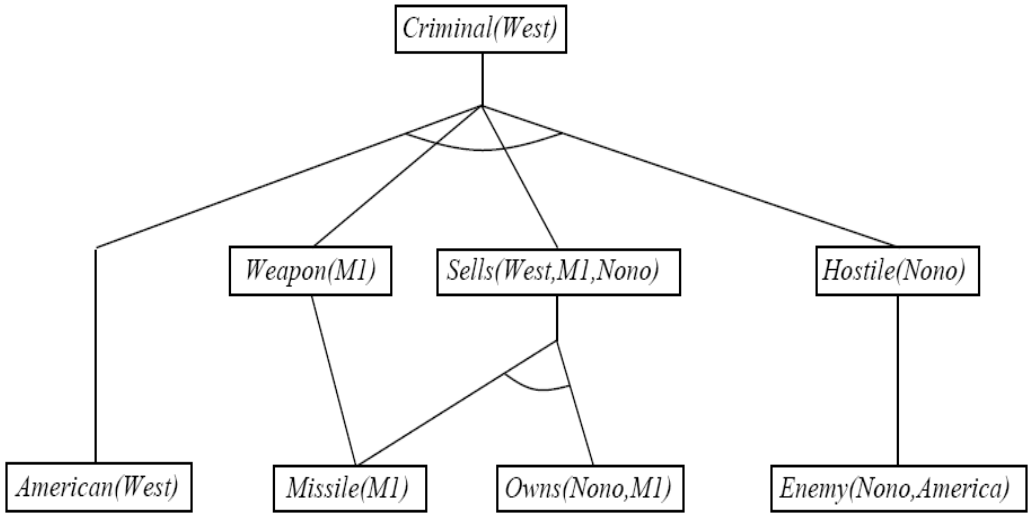


Figure 3.10 Example for forward Chaining

A simple forward-chaining algorithm, Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered or no new facts are added. A fact is not "new" if it is just a renaming of a known fact.

Algorithm:

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  repeat until new is empty
    new ← { }
    for each sentence r in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(r)
      for each θ such that (p1 ∧ ... ∧ pn)θ = (p'1 ∧ ... ∧ p'n)θ
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' is not a renaming of a sentence already in KB or new then do
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

Backward chaining

Backward chaining (or **backward reasoning**) is an inference method that can be described (in lay terms) as working backward from the goal(s). It is used in automated theorem provers, inference engines, proof assistants and other artificial intelligence applications.

Backward chaining starts with a list of goals (or a hypothesis) and works backwards from the consequent to the antecedent to see if there is data available that will support any of these consequents.[3] An inference engine using backward chaining would search the inference rules until it finds one which has a consequent (**Then** clause) that matches a desired goal. If the antecedent (**If** clause) of that rule is not known to be true, then it is added to the list of goals (in order for one's goal to be confirmed one must also provide data that confirms this new rule).

For example, suppose a new pet, Fritz, is delivered in an opaque box along with two facts about Fritz:

- Fritz croaks
- Fritz eats flies

The goal is to decide whether Fritz is green, based on a rule base containing the following four rules:

1. **If** X croaks and X eats flies – **Then** X is a frog
2. **If** X chirps and X sings – **Then** X is a canary
3. **If** X is a frog – **Then** X is green
4. **If** X is a canary – **Then** X is yellow

With backward reasoning, an inference engine can determine whether Fritz is green in four steps. To start, the query is phrased as a goal assertion that is to be proved: "Fritz is green".

1. Fritz is substituted for X in rule #3 to see if its consequent matches the goal, so rule #3 becomes:

If Fritz is a frog – **Then** Fritz is green

Since the consequent matches the goal ("Fritz is green"),the rules engine now needs to see if the antecedent ("If Fritz is a frog") can be proved. The antecedent therefore becomes the new goal:

Fritz is a frog

2. Again substituting Fritz for X, rule #1 becomes:

If Fritz croaks and Fritz eats flies – **Then** Fritz is a frog

Since the consequent matches the current goal ("Fritz is a frog"), the inference engine now needs to see if the antecedent ("If Fritz croaks and eats flies") can be proved. The antecedent therefore becomes the new goal:

Fritz croaks and Fritz eats flies

3. Since this goal is a conjunction of two statements, the inference engine breaks it into two sub-goals, both of which must be proved:

Fritz croaks

Fritz eats flies

4. To prove both of these sub-goals, the inference engine sees that both of these sub-goals were given as initial facts. Therefore, the conjunction is true:

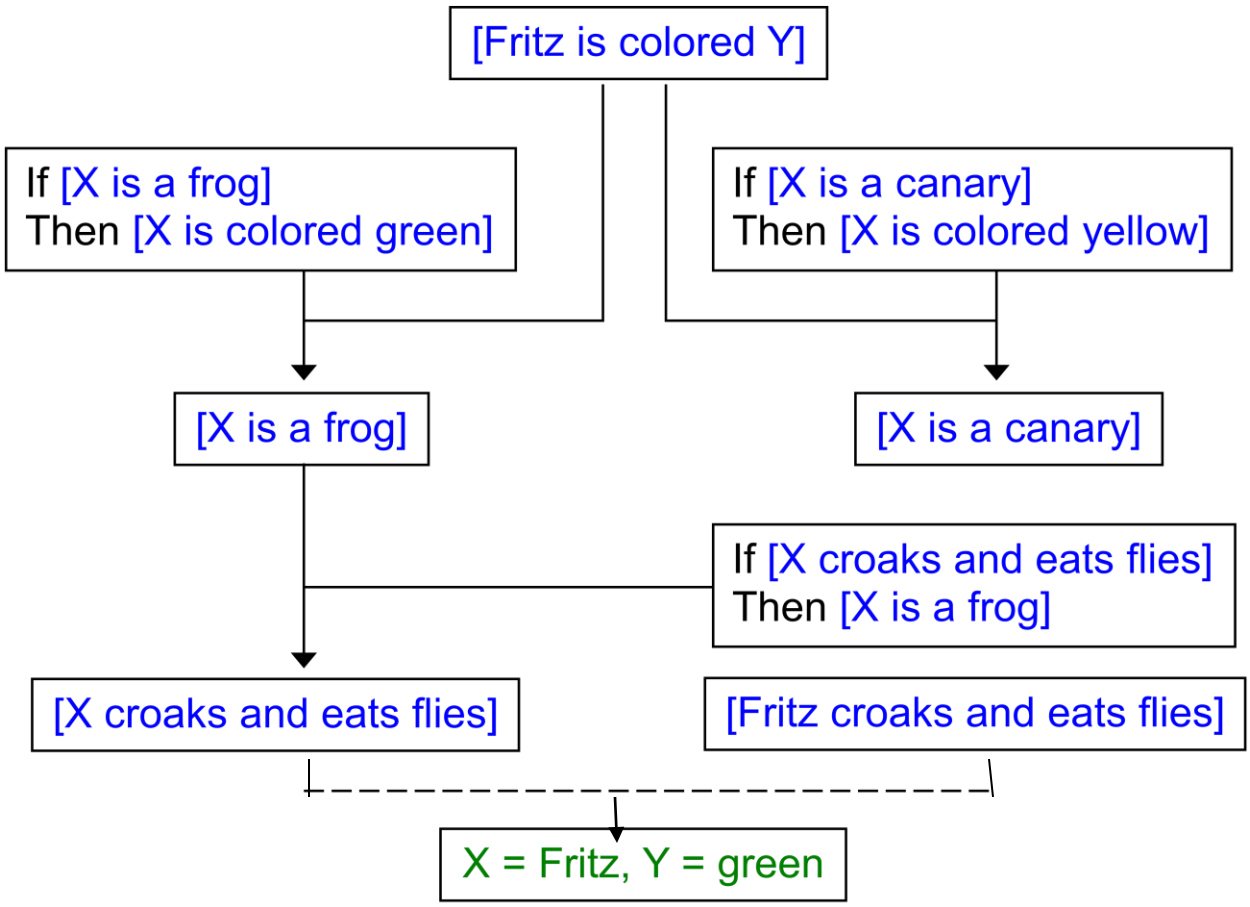
Fritz croaks and Fritz eats flies therefore the antecedent of rule #1 is true and the consequent must be true:

Fritz is a frog

Therefore the antecedent of rule #3 is true and the consequent must be true:

Fritz is green

This derivation therefore allows the inference engine to prove that Fritz is green. Rules #2 and #4 were not used.



Difference

Backward chaining (a la Prolog) is more like finding what initial conditions form a path to your goal. At a very basic level it is a backward search from your goal to find conditions that will fulfil it.

Backward chaining is used for interrogative applications (finding items that fulfil certain criteria) - one commercial example of a backward chaining application might be finding which insurance policies are covered by a particular reinsurance contract.

Forward chaining (a la CLIPS) matches conditions and then generates inferences from those conditions. These conditions can in turn match other rules. Basically, this takes a set of initial conditions and then draws all inferences it can from those conditions.

Algorithm

function FOL-BC-ASK(KB, goals, θ) **returns** a set of substitutions
  inputs: KB, a knowledge base
    $goals$, a list of conjuncts forming a query
    θ, the current substitution, initially the empty substitution $\{\}$
  local variables: ans, a set of substitutions, initially empty

  if $goals$ is empty **then return** $\{\theta\}$
  $q' \leftarrow$ SUBST(θ, FIRST($goals$))
  for each r **in** KB where STANDARDIZE-APART(r) $= (p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
    and $\theta' \leftarrow$ UNIFY(q, q') succeeds
   $ans \leftarrow$ FOL-BC-ASK(KB, $[p_1, \ldots, p_n | \text{REST}(goals)]$, COMPOSE($\theta', \theta$)) $\cup$ ans
  return ans

RESOLUTION

In 1930, the German mathematician Kurt Godel proved the first completeness theorem for first-order logic, showing that any entailed sentence has a finite proof. In 1931, Godel proved an even more famous incompleteness theorem.

- The theorem states that a logical system that includes the principle of induction without which very little of discrete mathematics can be constructed-is necessarily incomplete. Hence, there are sentences that are entailed, but have no finite proof within the system.
- Resolution-based theorem provers have been applied widely to derive mathematical theorems, including several for which no proof was known previously. Theorem provers have also been used to verify hardware designs and to generate logically correct programs, among other applications like
  - Conjunctive normal form for first-order logic.
  - The resolution inference rule

- Completeness of Resolution
- Dealing with equality
- Resolution strategies
- Theorem provers

Conjunctive normal form for first-order logic

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

$\forall x$American(x) $\wedge$ Weapon(y))$\wedge$ Sells(x, y, z) $\wedge$Hostile(z) $\Rightarrow$ Criminal(x)

Becomes in CNF,

¬American(x) V ¬Weapon(y))V¬ Sells(x, y, z) V¬Hostile(z) V Criminal(x)

Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. The procedure for conversion to CNF is very similar to the propositional case.

Eg: **Everyone who loves all animals is loved by someone**

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

The steps are as follows:

1. **Eliminatebiconditionals and implications:**

   $\forall x$ [¬$\forall y$¬ Animal(y) V Loves(x,y)] V[$\exists y$ Loves(y,x)]

2.Move ¬ inwards:

   ¬$\forall x$ p becomes $\exists x$ ¬p

   ¬$\exists x$ p becomes $\forall x$ ¬p

The transforms are

   $\forall x$ [$\exists y$¬(¬ Animal(y) V Loves(x,y))] V[$\exists y$ Loves(y,x)]

   $\forall x$ [$\exists y$¬¬ Animal(y) $\wedge$¬ Loves(x,y)] V[$\exists y$ Loves(y,x)]

   $\forall x$ [$\exists y$ Animal(y) $\wedge$¬ Loves(x,y)] V[$\exists y$ Loves(y,x)]

3. Standardize variables: For sentences like (Vx P(x)) V (:3x 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 each quantifier should use a different one

   $\forall x$ [$\exists y$ Animal(y) $\wedge$¬ Loves(x,y)] V[$\exists z$ Loves(z,x)]

4. Skolemize: a more general form of existential instantiation. Skolemization is the process of removing existential quantifiers by elimination. Each existential variable is replaced by a Skolem function of the enclosing universally quantified variables:

$$\forall x\ [\ Animal(A) \wedge \neg\ Loves(x,A)]\ V\ Loves(B,x)$$

5. 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.

$$Animal(F(x)) \wedge \neg\ Loves(x,F(x))]\ V\ Loves(G(x),x)$$

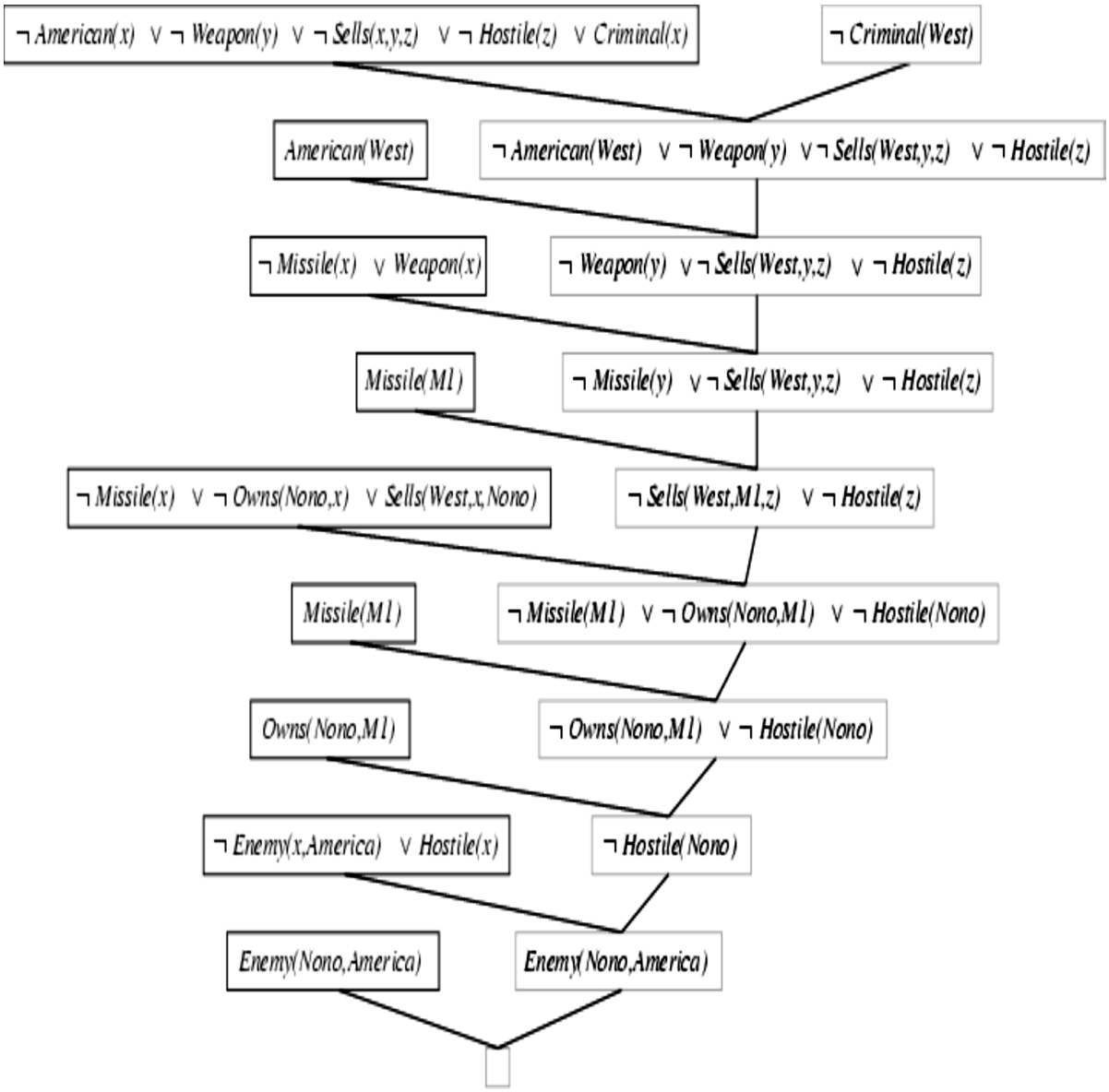
6.**Distribute $\wedge$ over V:**

$$[Animal(F(x))\ V\ \ Loves(G(x),x)] \wedge [\ \neg Loves(x,F(x))VLoves(G(x),x)]$$

This step may also require flattening out nested conjunctions and disjunctions. The sentence is now in CNF and consists of two clauses.

1. F(x) refers to the animal potentially unloved by x
2. G(x) refers to someone who might love x

Example



Example

This makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English, the problem is as follows:

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

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:

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

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

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

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. $\neg Kills(Curiosity, Tuna)$

Example:

1.Jack owns a dog.

2.Every dog owner is an animal lover.

3.No animal lover kills an animal.

4.Either Jack or Curiosity killed the cat, who is named Tuna.

5.Did Curiosity kill the cat?

1. $\exists x : Dog(x) \wedge Owns(Jack, x)$

2. $\forall x; \quad (\exists y \quad Dog(y) \wedge Owns(x, y)) \rightarrow AnimalLover(x)$

3. $\forall x; \quad AnimalLover(x) \rightarrow (\forall y \quad Animal(y) \rightarrow \neg Kills(x, y))$

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

5. $Cat(Tuna)$

6. $\forall x : \quad Cat(x) \rightarrow Animal(x)$

Conjunctive Normal Form

$Dog(D)$

$Owns(Jack, D)$

$\neg Dog(y) \vee \neg Owns(x, y) \vee AnimalLover(x)$

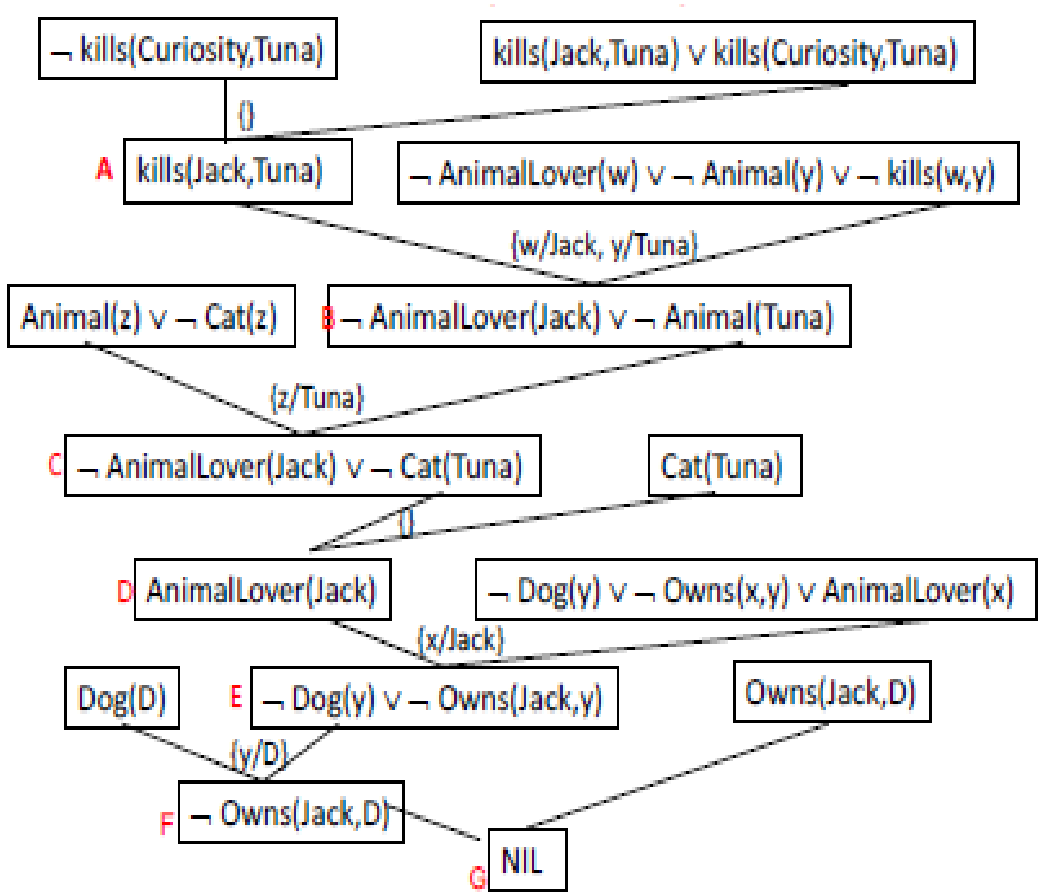
$\neg AnimalLover(w) \vee \neg Animal(y) \vee \neg Kills(w, y)$

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

$Cat(Tuna)$

$\neg Cat(z) \vee Animal(z)$

$\neg Kills(Curiosity, Tuna)$



Resolution Strategies

Unit resolution: prefer to perform resolution if one clause is just a literal yields shorter sentences.

•**Set of support** : identify a subset of the KB (hopefully small); every resolution will take a clause from the set and resolve it with another sentence, then add the result to the set of support.

•**Input resolution**: always combine a sentence from the query or KB with another sentences.

Linear resolution: resolve P and Q if P is in the original KB or is an ancestor of Q in the proof tree.

• **Subsumption**: eliminate all sentences more specific than a sentence already in the KB

Demodulation

Demodulation is typically used for simplifying expressions using collections of assertions such as $x + 0 = x$, $x1 = x$, and so on. The rule can also be extended to handle non-unit clauses in which an equality literal appears:

Paramodulation

Unlike demodulation, paramodulation yields a complete inference procedure for first-order logic with equality.

TRUTH MAINTENANCE SYSTEM

Many of the inferences drawn by a knowledge representation system will have only default status, rather than being absolutely certain. Inevitably, some of these inferred facts will turn out to be wrong and will have to be retracted in the face of new information. This process is called **belief revision**.

Truth Maintenance Systems (TMS) have been developed as a means of implementing Non-Monotonic Reasoning Systems.

Basically TMSs:

- all do some form of dependency directed backtracking
- assertions are connected via a network of dependencies.

Justification-Based Truth Maintenance Systems (JTMS)

- This is a simple TMS in that it does not know anything about the structure of the assertions themselves.
- Each supported belief (assertion) in has a justification.
- Each justification has two parts:
    - An IN-List -- which supports beliefs held.
    - An OUT-List -- which supports beliefs not held.
- An assertion is connected to its justification by an arrow.
- One assertion can feed another justification thus creating the network.
- Assertions may be labelled with a belief status.
- An assertion is valid if every assertion in the IN-List is believed and none in the OUT-List are believed.
- An assertion is non-monotonic is the OUT-List is not empty or if any assertion in the IN-List is non-monotonic.

Logic-Based Truth Maintenance Systems (LTMS)

Similar to JTMS except:

- Nodes (assertions) assume no relationships among them except ones explicitly stated in justifications.
- JTMS can represent P and ¬P simultaneously. An LTMS would throw a contradiction here.
- If this happens network has to be reconstructed.

Assumption-Based Truth Maintenance Systems (ATMS)

- JTMS and LTMS pursue a single line of reasoning at a time and backtrack (dependency-directed) when needed -- depth first search.
- ATMS maintain alternative paths in parallel -- breadth-first search
- Backtracking is avoided at the expense of maintaining multiple contexts.
- However as reasoning proceeds contradictions arise and the ATMS can be pruned
    - Simply find assertion with no valid justification.