First order Logic [FOL]

Representation Revisited:

→ Programming Languages (like C++/Java/LISP) are the largest class of formal languages which are in common use to represent computational processes.

→ The Data structures with in the programs represents "facts".

for Ex: Consider 4x4 array to represent wumpus world contents.

i.e, world [2,2] ← pit.

→ it's a way to assert that there is a pit in square [2,2].

→ These languages are lack in deriving facts from other facts. Each update to a data structure is done by domain specific procedure, whose details are derived by the programmer from his/her own knowledge of domain.

→ This procedural approach can be contrasted with declarative nature of propositional logic in which

knowledge & inference are separate.
[inference is entirely domain independent]

→ Another drawback of datastructures in programs is lack of easy way to say.

for ex: There is a pit in [2,2] (or) [3,1]

Programs can store a single value for each variable and some systems allow the value to be "unknown". ~~progress~~ ~~can~~ ~~store~~ But they lack in expressiveness required to handle partial information.

→ PL is declarative Language because its semantics is based on truth relation between sentences & words.

→ It has expressive power to deal with Partial information using disjunction & ¬

→ main property of PL is "Compositionality". Meaning of a sentence is a function of meaning of its parts.

for ex: $S_{1,4} \wedge S_{1,2}$ is related to meanings of $S_{1,4}$ and $S_{1,2}$.

→ Drawback of PL is - Lacks the expressive power to describe an environment

with many objects Concisely.

for ex: $B_{1,1} \iff (P_{1,2} \vee P_{2,1})$

Here, we were forced to write a separate rule about breezes and pits for each square.

→ But if we Consider English it seems easy to say once for all i.e, "Squares adjacent to pits are Breezy"

→ So, Natural Language is a declarative knowledge representation language.

So, the syntax of natural langue Consists of nouns & noun phrases that refer to objects [squares, pits, wumpus] and verbs, verb phrases refer to relations among objects [breezy, is adjacent to, shoots]

→ Some of these relations are functions in which there is only one value for a given I/P.

objects: people, houses, numbers, colors, ...

Relations: there can be unary relations (or) properties

i.e, red, round, prime...

Can more general n-ary relations like brother of, bigger than...

Functions : father of, best friend....

① Ex : one plus two equals three"

obj's : one, two, three, one plus two

Relation : equals

Function : "plus"

[it is the name for object that is obtained by applying the function plus to objects one and two

∴ three is another name for this object]

② Ex : "squares neighboring the wumpus are smelly"

obj's : wumpus, squares

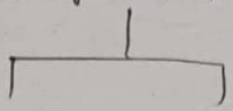
Relation : neighboring

function/property : smelly.

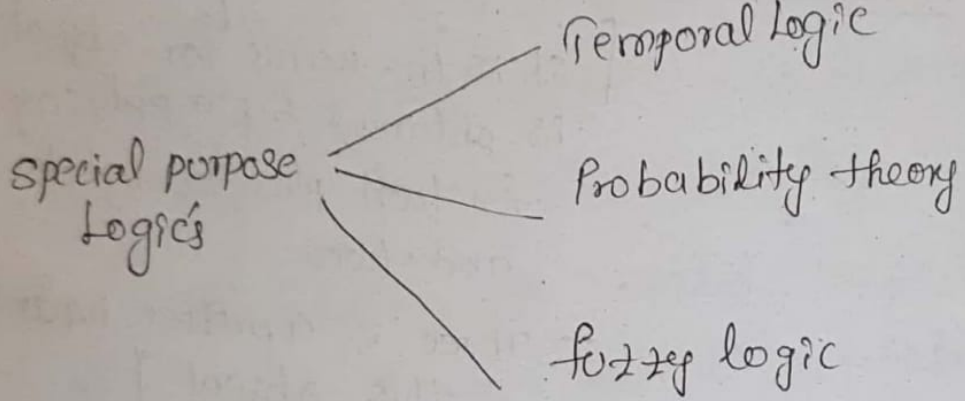
→ The major difference between PL and FOL lies in ontological Commitment made by language. i.e, what it assumes about nature of reality.

forEX: PL assumes that there are facts that either hold (or) do not hold in the world.

Each fact can be in one of two states



But, FOL assumes more i.e, objects and its relations among them that do or don't hold.



A logic can, also be characterized by its epistemological Commitments. i.e, the possible states of knowledge that it allows with respect to each fact.

Language	ontological Commitment (what exists in the world)	Epistemological commitment (what agent believes about facts)
PL	facts	true/false/unknown
FOL	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 of FOL

Models for FOL:

→ As we know, models for PL are just sets of truth values for the proposition symbols.

→ Models for FOL are;

The Domain of the model is set of objects it contains, and these are called domain elements.

→ Here is the model with 5 objects. i.e, two binary relations, three unary relations. i.e,

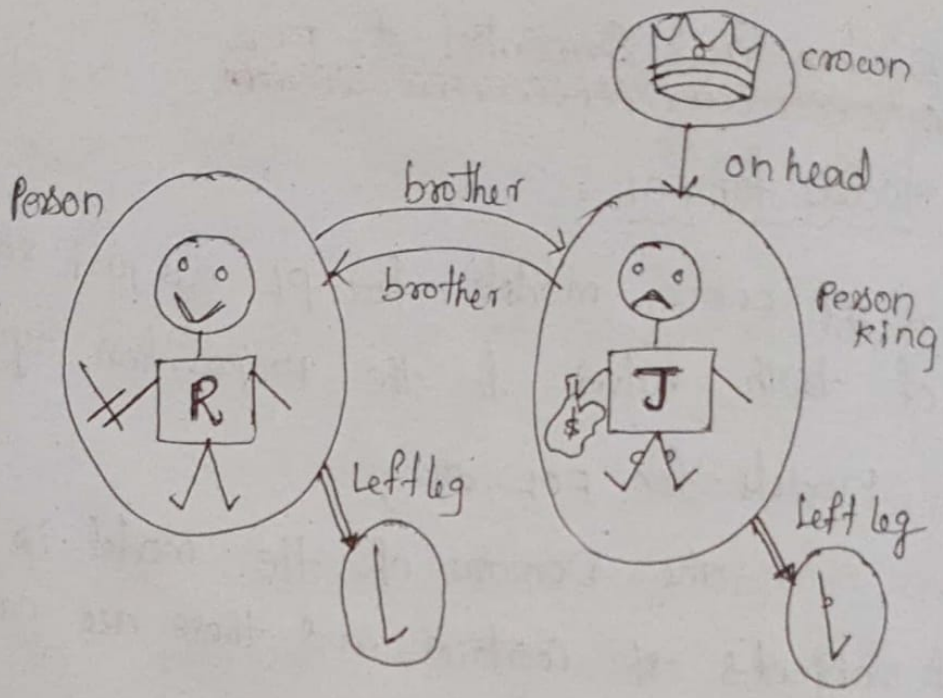
Objects: ✓ Richard
✓ John
✓ Left legs of R&J
✓ ~~Younger brother~~ crown
✓ Brother

Binary relation

→ These objects may be related in many ways. i.e, Richard & John are brothers in figure. so, a relation is set of tuples of objects that are related.

Thus, brotherhood relation in this model is the set.

$$\{(R, J), (J, R)\}.$$



→ Here the crown is on King John's head. So the "on head" relation Contains just one tuple. (the crown, King John)

→ The "brother" and "on head" relations are binary relations. i.e, they relate pairs of objects.

→ This model Contains unary relations (or Properties: "person" → True for both R & J
"King" → True for only J
"Crown" → True for only crown

→ Some kinds of relationships are best Considered as functions. In that object must be related exactly one object in this way.

forex: each person has one left leg so model has unary "left leg"

function which includes the following:

< Richard > → Richard's leftleg

< John > → John's left leg.

symbols & interpretations:

sentence → Atomic Sentence
| (sentence Connective Sentence)
| (Quantifier . variable, ... sentence)
| ¬ sentence.

Atomic sentence → predicate(Term, ...) | Term = Term

Term → Function(Term, ...)
| Constant
| variable

Connective → ⇒ | ∧ | ∨ | ⇔

Quantifier → ∀ | ∃

Constant → A | X1 | John | ...

variable → a | x | s | ...

Predicate → before | Hascolor | Raining | ...

Function → mother | leftleg | ...

→ This is the syntax of FOL which is equally specified in BACKUS-NAUR FORM.

Basic syntactic elements of FOL are;

→ symbols that stands for objects, relations, functions.

→ symbols of 3 kinds :

i) Constant symbols
stand for objects

ii) Predicate symbols
stand for relations

iii) Function symbols
stand for functions.

→ Each predicate & function symbol comes with an arity that fixes the number of arguments.

→ In order to determine the truth that relate sentences to models we need Interpretation. which specifies exactly which objects, relations and functions are referred to by the Constant, Predicate, and function symbols.

Terms:

→ A logical expression that refers to an object

→ Constant symbols are terms

→ A complex term is formed by a function symbol followed by parantheized list of terms as arguments.

$f(t_1, \cdots t_n)$.

↳ term refers to objects in domain $(d_1, \cdots d_n)$

Eg:

King John's left leg" referred as

Leftleg(John).

Atomic Sentences:

Terms for referring to objects & predicate symbols for refferring to relations can together form "Atomic Sentence".

→ it is formed with predicate symbol followed by paranthesised list of terms.

i.e, Brother (Richard, John)

→ Atomic Sentences can have Complex Terms as arguments.

married (Father (Richard) Mother(John))

So, it is true in a given model iff the relation referred by predicate symbol

holds among the objects.

Complex sentences:

It uses Logical Connectives

i.e,

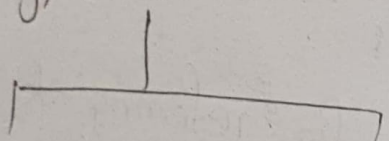
¬ Brother (leftleg (Richard), John)

Brother (Richard, John) ∧ Brother (John, Richard)

Quantifiers:

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

Two types of Quantifiers



Universal (∀)

Existential (∃)

① Universal: All Kings are persons

∀x King(x) ⇒ person(x)

i.e, forall x, if x is a King then x is a person.

[x: is available / act as argument of function]

Ex: leftleg(x)

Ground term : A Term with no variables is called a ground term.

Sentence $\forall x\ P$

P is true for every object x

[where P is a logical expression]

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

i.e,
- $x \longrightarrow$ Richard the lion heart
- $x \rightarrow$ King John
- $x \rightarrow$ Richard's left leg
- $x \rightarrow$ John's left leg
- $x \rightarrow$ the crown.

i.e, $[\forall x \quad King(x) \Rightarrow person(x)]$ is true iff

$King(x) \Rightarrow person(x)$ is true

in each of five extended interpretations.

① Richard the Lion heart is a king => Richard the lion heart is a person

② King John is a king => King John is a person

③ Richard's left leg is a king => Richard's left leg is a person.

④ John's left leg is a king ⇒ John's left leg is a person

⑤ The crown is a king ⇒ the crown is a person.

But, ~~if we observe~~ the above sentences we know king John is only king. and remaining four sentences about to be claim.

But we can use Conjunction instead of implication.

$\forall x$ king $(x) \wedge$ person (x)

will assert the above sentences in a different way.

② Existential

Ex: $\exists x$ Crown $(x) \wedge$ OnHead $(x,$ John$)$

King John has crown on his head.

The sentence $\exists x\ P$ says that P is true for atleast one object x. i.e, one of the following must be True.

Richard the Lionheart is crown $\wedge$ Richard the lionheart is on John's head

King John is a crown $\wedge$ King John is on John's head

✓ The crown is a Crown $\wedge$ Crown is on John's head

So, the Sentence is true in the model.

Nested quantifiers :

To express more Complex Sentences using multiple quantifiers. The simplest case is when the quantifiers are of same type.

Ex : "Brothers are siblebings"

$$\forall x \forall y \text{ Brother}(x,y) \Rightarrow \text{sibling}(x,y)$$

⇒ quantifiers of same type can be written as one quantifier with several variables

for ex : sibling is a symmetric relationship

$$\forall x,y \text{ Sibling}(x,y) \Leftrightarrow \text{sibling}(y,x)$$

other cases ;

Ex : "Everybody loves somebody" } mixtures

$$\forall x \exists y \text{ Loves } (x,y)$$

(or)

Ex : "There is somebody who is loved by everyone"

$$\exists y \forall x \text{ Loves}(x,y).$$

Ex : "Here is some query on which Confusion can arise when two quantifiers are used with same variable name.

$$\forall x \; [\text{Crown}(x) \vee (\exists x \text{ Brother}(\text{Richard}, x))]$$

Connectives between quantifiers:

Two quantifiers are actually connected with each other, though negation

Everyone dislikes chocolate is same as saying there does not exist someone who likes them.

$\forall x \neg Likes(x, Chocolate)$ is equal to $\neg \exists x Likes(x, chocolate)$

Equality:

Equality symbol is used to make statements to the effect that two terms refer to the same same object.

for ex: father(John) = Henry

i.e, the object referred by father(John) and the object referred by Henry are same

→ Equality symbol is used to state facts about given function.

→ It also be used with negation to mean that two terms are not same object. for ex: Richard has atleast 2 brothers

$\exists x, y$ Brother $(x, \text{Richard}) \wedge$ Brother $(y, \text{Richard}) \wedge \neg(x, y)$

Using FOL:

Here we focus on more systematic representations of some simple domains.

→ In KB (or knowledge Representation, a domain is just some part of world abt which we wish to express some knowledge.

Assertions and queries in FOL:

As we know that, sentences are added to a KB using "TELL"; exactly as like in PL. such sentences are called assertions.

forEx: assert [John is a King and that Kings are persons:

TELL (KB, King (John))
TELL (KB, $\forall x$ King $(x) \Rightarrow$ person (x))

we can ASK a question of KB using "ASK"

forEx: ASK (KB, King (John))

which returns TRUE.

So, questions asked using ASK are called queries (or goals)

⇒ Suppose. Consider a query;

Ask (KB, person (John)).

It returns true.

⇒ Suppose if we have quantified queries like,

Ask (KB, ∃x, person(x)).

It is also true But, here we have existential variable is asking "Is there an x such that" so we solve it by providing x. so this the type of procedure can be used solve this problem is substitution / binding list.

The Answer is, {x/John}.

The Kinship domain:

for example Consider the domain of family relationships. It includes

"Elizabeth is the mother of charles"

and

"charles is father of william"

and rules like;

"one's grandmother is mother of one's parent"

→ The objects in our domain are people.

We have two unary predicates { male, Female

→ Kinship relations — parenthood, brotherhood, marriage and so on ...

→ Binary predicates — parent, sibling, Brother, sister, child, daughter, son, spouse, wife, Husband, Grand parent, Grandchild, Cousin, Aunt and uncle.

→ The functions we use are { mother, Father

So, now Consider; "one's mother is one's female parent"

$\forall m, c \cdot Mother(c) = m \iff Female(m) \wedge Parent(m, c)$

"one's husband is one's male spouse"

$\forall w, h \quad Husband(h, w) \iff male(h) \wedge Spouse(h, w)$

"male and female are disjoint Categories":

$\forall x \quad Male(x) \iff \neg Female(x)$.

"Parent and child are inverse relations"

$\forall p, c \quad parent(p, c) \iff child(c, p)$.

"A Grand Parent is a parent of one's parent"

$\forall g, c \quad Grandparent(g, c) \iff \exists p \; Parent(g, p) \wedge Parent(p, c)$.

"A sibling is another child of one's parents"

$\forall x,y \; Sibling(x,y) \Leftrightarrow x \neq y \wedge \exists p \; Parent(p,x) \wedge Parent(p,y)$.

→ Each of these sentences can be viewed as an axiom of kinship domain.

They provide basic fundamentals from which useful conclusions can be derived. These are also called definitions which are of the form

$\forall x,y \; P(x,y) \Leftrightarrow \ldots$

Theorems:

Not all logical sentences about a domain are axioms. Some are theorems --- i.e, they entailed by the axioms. for ex:

Consider assert that siblinghood is symmetric

$\forall x,y \; Sibling(x,y) \Leftrightarrow Sibling(y,x)$

which returns true by KB.

Numbers, sets and lists:

Numbers: Here consider Natural numbers they will be true for natural numbers, and we need a constant symbol 0, and one function symbol S (successor) if Natural numbers are defined recursively

$NatNum(0)$

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

so natural numbers are 0, S(0), S(S(0)) - - -

Successor function;

$\forall n \; 0 \neq S(n)$

$\forall m, n \; m \neq n \Rightarrow S(m) \neq S(n)$.

Now, we can define addition in terms of S

$\forall m \; NatNum(m) \Rightarrow +(m,0) = m$

$\forall m, n \; NatNum(m) \wedge NatNum(n) \Rightarrow$

$+(S(m), n) = S(+(m,n))$.

This can be considered in INFIX as;

$\forall m, n \; NatNum(m) \wedge NatNum(n) \Rightarrow$

$(m+1)+n = (m+n)+1$.

SETS :

1) Sets are empty and those made by adjoining something to a set.

$\forall s \; set(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2 \; set(s_2) \wedge s = \{x/s_2\})$

2) If empty set has no elements.

$\neg \exists x, s \; \{x/s\} = \{\}$.

3) Adjoining an element already in set.

$\forall x, s \; x \in s \Leftrightarrow s = \{x/s\}$

4) x is a member of s if and only if s is equal to some set s_2 by adjoined with some element y.

$\forall x, s \; x \in s \Leftrightarrow [\exists y, s_2 \; (s = \{y/s_2\} \wedge (x = y \vee x \in s_2))]$.

5) A set is a subset of another set

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

6) Two sets are equal iff each is a subset of other $\forall s_1, s_2 \; (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1)$

7) Intersection of sets

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

8) Union of sets

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

Lists : similar to sets but ...
appear more than once in a list.
functions : Cons, Append, First, Rest and so on...
Cons(x, Nil) → [x].

---

wumpus world :
Recall wumpus receives percepts with 5 elements
percept([Stench, Breeze, Glitter, None, None], 5)
→ actions represented as;
Turn(Right), Turn(left), Forward, shoot, Grab, Climb
To determine best action Agent Construct a query
∃a BestAction(a, 5).
ASK will solve this query as {a/Grab}
Before this, it must TELL to its own KB that performing Grab.
∀t, s, g, m, c percept([s, Breeze, g, m, c], t) ⇒ Breeze(t)
∀t, s, b, m, c percept([s, b, Glitter, m, c], t) ⇒ Glitter(t)
i.e, [∀t | Glitter(t) ⇒ BestAction(Grab, t).]

Diagnostic rules :
i.e, Lead from observed effects to hidden causes
∀s Breezy(s) ⇒ ∃r Adjacent(r, s) ∧ pit(r)
and also,
∀s ¬Breezy(s) ⇒ ¬∃r Adjacent(r, s) ∧ pit(r)
By combining these two we have,
[∀s Breezy(s) ⇔ ∃r Adjacent(r, s) ∧ pit(r)]

Casual rules :
i.e, hidden property of world causes a percept
∀r pit(r) ⇒ [∀s Adjacent(r, s) ⇒ Breezy(s)]
if all squares adjacent to given square is pitless then it will not breezy.
∀s [∀r Adjacent(r, s) ⇒ ¬pit(r)] ⇒ ¬Breezy(s)

Inference in FOL

① Propositional vs first-order inference.:

Here we consider some simple inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers.

Inference rules for quantifiers:

① Let us consider universal quantifiers suppose our KB contains the standard axiom states that all greedy kings are evil:

$$\forall x \; King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$

Then, it is permissible to infer any of the following sentences.

$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$

$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$

$King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John))$

⋮

The rule of universal instantiation says that we can infer any sentence obtained by substituting a ground term for the variable.

To write inference rule;

Let SUBST(θ, α) denote the result of applying the substitution θ to the sentence α then the rule can be written as,

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

i.e, for any variable 'v', and ground term 'g' the substitutions can be;

{x/ John}
{x/ Richard}
{x/ Father(John)}

(2) The "existential Instantiation" rule for the existential quantifier is slightly more complicated. i.e,

for any sentence α, variable v and constant symbol k that doesn't appear elsewhere in KB

the rule can be written as;

$$\frac{\exists v\ \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

for example; from the sentence.

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

we can infer the sentence.

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

i.e, C_1 doesn't appear elsewhere in knowledge base.

→ Here, the universal instantiation, can be applied many times to produce many Consequences. But, Existential instantiation Can applied once. Hence, this can be discarded.

forex: we added sentence

Kill (murderer, victim)

so we no longer need the sentence

$\exists x$ Kill $(x,$ victim$)$.

→ so new knowledge base is inferentially equivalent but not logically equivalent to old one.

Reduction to propositional inference:

Suppose KB Contains the following:

KB

$\forall x$ King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x)
King(John)
Greedy(John)
Brother(Richard, John)

→ Now, apply universal instantiation (UI) using all possible ground term substitutions i.e, {x/John} {x/Richard}

we obtain;

King(John) $\wedge$ Greedy(John) $\Rightarrow$ Evil(John)
King(Richard) $\wedge$ Greedy(Richard) $\Rightarrow$ Evil(Richard)

new KB

and
King(John)
Greedy(John)
Brother(John, Richard)

↓ propositional symbols.

If we apply an propositional Algorithm it will Conclude Evil(John).

→ Every FOL KB Can be propositionalized so as to preserve entailment.

i.e, "A grand sentence Can be entailed by new KB iff it is entailed by original KB"

So, the idea for doing inference in FOL is,

→ Propositionalize KB and query
→ Apply resolution based inference
→ Return Result

There is a problem with functionsymbols i.e, if there are many ground terms like,

father (father (father (John))), ...

Theorem : Herbrand (1930). if a sentence α is entailed by a FOL KB, it is entailed by finite subset of the propositionalized KB.

we can find subset by first generating all instantiations with Constant symbols (Richard and John) then all terms of depth 1 (father (Richard) and father (John))

and so on ---

→ This can be Continued until the proof is done that the sentence is entailed.

(2) Unification and Lifting:

Propositionalization approach is inefficient because it generates a lot of irrelevant sentences. i.e,

from query Evil(x) and KB

it generate sentences { King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)

Evil(John)

from the knowledgebase;

∀x King(x) ∧ Greedy(x) ⇒ Evil(x)

King(John)

Greedy(John)

First order inference rule:

The inference that John is evil works like;

Find some x such that x is a king and x is greedy then infer x is evil.

i.e, if there is a substitution θ that makes the premise of implication identical to sentences already in KB. then, we can assert the conclusion {x/John}.

Suppose we know that, instead of knowing Greedy (John) we know everyone is greedy.

$\forall y$ (Greedy (y))

then we still Conclude it to Evil (John) because John is King and John is greedy

ie, $\theta = \{x/John, y/John\}$

So if we apply these substitutions to implication premises King (x), greedy (x) then the knowledgebase sentences are identical.

King (John) and Greedy (y)

Generalized modus ponens:

This a general inference rule for FOL that doesnot require instantiation.

ie, for atomic sentences P_i, P_i' and q where there is substitution θ such that,

$$SUBST(\theta, P_i') = SUBST(\theta, P_i), \text{ for all } i.$$

$$\frac{P_1', P_2', \ldots, P_n', (P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow q)}{SUBST(\theta, q)}$$

Here there are n+1 premises to this rule.

i.e,

a atomic sentences p_i' and one implication.
the conclusion by substitution of θ to Consequent q. i.e,

p_1' is King(John)
p_2' is Greedy(y)
θ is {x/John, y/John}

p_1 is King(x)
p_2 is Greedy(x)
q is Evil(x)

SUBST(θ, q) is Evil(John).

GMP is a sound inference rule. i.e, for a θ that satisfies any sentence p

$$P \vdash SUBST(\theta, P)$$

→ Gmp Can be lifted from PL to FOL the key advantage is that they Make only those substitutions which are allow particular inferences to proceed.

The key Component of first order inference algorithm is unification :

Unification :
Lifted inference rules require finding substitutions that make different logical expressions look identical.

The UNIFY algorithm takes 2 sentences and returns a unifier for them if one exists:

UNIFY $(P, Q) = \theta$ where SUBST(θ, P) = SUBST(θ, Q)

→ Suppose a query knows(John, x)

→ Here are some results of unification from KB,

UNIFY (knows (John, x), knows (John, Jane)) = {x/Jane}.

UNIFY (knows (John, x), knows (y, Bill)) = {x/Bill, y/John}.

UNIFY (knows (John, x), knows (y, mother(y)) = {y/John, x/mother(John)}

UNIFY (knows (John, x), knows (x, Elizabeth)) = fail

The last unification fails because x can't take values John & Elizabeth at same time.

Consider, knows (x, Elizabeth).

⇓

"means Everyone knows Elizabeth"

⇓, means

"John knows elizabeth"

→ This problem can be avoided by "standardizing apart"

i-e, renaming its variables to avoid name clashes.

i-e, for Ex: x in knows(x, Elizabeth) is renamed as z_{17}

UNIFY(knows(John, x), knows(z_{17}, Elizabeth)) =
{x/Elizabeth, z_{17}/John}

→ the unification algorithm is required to return the (unifier) most general unifier

i-e, fewer restrictions can be made by unifier.

unify(knows(John, x), knows(y, z))
returns {y/John, z/x}
(or)
{y/John, x/John, z/John}

Algorithm:

Function UNIFY(x, y, θ) returns a substitution to make x and y identical.

inputs: x, a variable, constant, list or Compound expression
y, a variable, Constant, list or Compound expression
θ, substitution built up so far (optional, default empty)

If θ = failure then return failure
else if x = y then return θ
else if variable?(x) then return UNIFY-VAR(x, y, θ)
else if variable?(y) then return UNIFY-VAR(y, x, θ)
else if COMPOUND?(x) and COMPOUND?(y) then

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

---

function UNIFY-VAR(var, x, θ) returns substitution
if {var/val} ∈ θ then return UNIFY(val, x, θ)
else if {x/val} ∈ θ then return UNIFY(var, val, θ)
else if OCCUR-CHECK?(var, x) then return failure.
else return add {var/x} to θ.

Storage & Retrieval :

Store → stores a sentence S into the knowledge base.

Fetch → returns all unifiers such that the query q unifies with some sentence in KB

The problem is; find all facts that unify with Knows(John, x) . by fetching

① The simplest way to implement these functions is to keep all facts in knowledge base in one long list.

② for a given query call Unify(q, s)

③ This process is inefficient, but it works.

these we can perform fetching more efficiently by avoiding some facts from knowledge base i.e,

Brother (Richard, John).

→ So, a simple scheme called "predicate indexing" puts all the known facts in one bucket and all Brothers facts in another. These buckets can be stored in hashtable for efficient access.

→ predicate indexing is useful when there are many ~~clauses for~~ predicate symbols but only few clauses for each symbol.

→ But in some cases, there are many clauses for a given predicate symbol. for example: "Tax authorities want to keep track of who employs whom" using a predicate Employs(x,y).

→ This will be a very large bucket with millions of employees.

→ Suppose we have a query Employs(x, Richard) it requires scanning the entire bucket.

→ So, it is helpful for getting a solution, if facts were indexed both by predicate and by second argument, using a combination of hash table key.

then it is simple to construct ~~query~~ key from query and retrieve exactly those facts that unify with query.

i.e, Employs (ALMA.org, Richard) is the fact and the queries are;

{ Employs (ALMA.org, Richard)
Employs (x, Richard)
Employs (ALMA.org, y)
Employs (x, y).

Subsumption Lattice:

This lattice has some intresting properties forex: The child of any node in the lattice is obtained from its parent by a single substitution and "highest common descendant" of any two nodes is the result of applying their most general unifier.

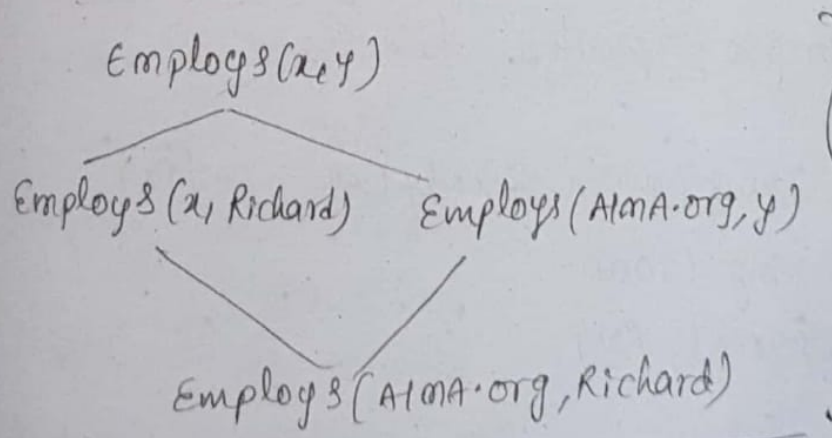


fig (a)

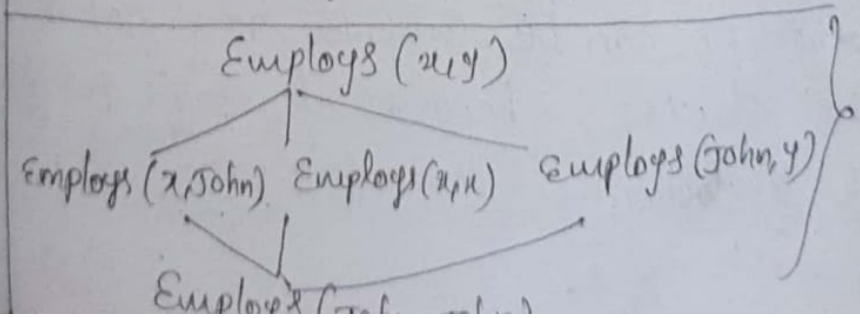


fig (b)

Forward chaining :

① start with atomic sentences in KB and apply modus ponens in forward direction. by adding new sentences until no further inferences can be made.

② 80 algorithm is applied to first-order definite clauses such as situation ⇒ Response

First order definite clauses :

it is closely resemble propositional definite clauses. They are disjunctions of literals ~~of~~ which exactly one is positive.

→ A definite clause is either atomic (or) ~~implication of whose antecedent is a conjunction of positive literals~~

An implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal.

i.e, $King(x) \wedge Greedy(x) \Rightarrow Evil(x)$
$King(John)$,
$Greedy(y)$.

[Not every KB can be converted in to set of definite clauses, because of single-positive literal restriction, but many can ..]

Consider the problem:

[The Law says that it is a crime for an American to sell weapons to hostile nations. The County Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel west; who is an American]

so, we need to prove that "west is a criminal".

(1) we need to represent these facts as first-order definite clauses.

"It is a crime for an American to sell weapons to hostile nations".

$American(x) \wedge weapon(y) \wedge sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$

(2) "Nono ... has some missiles".

$\exists x\ owns(Nono, x) \wedge missile(x)$ can be transformed to two definite clauses by Existential elimination. i.e,

$owns(Nono, M_1)$

$missile(M_1)$

(3) "All of its missiles were sold to it by colonel west"

$missile(x) \wedge owns(Nono, x) \Rightarrow sells(west, x, Nono)$.

(4) Enemy of America counts as "hostile"

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

(5) missiles are weapons

missile(x) ⇒ weapon(x)

(6) west, who is an American

American(west)

(7) "the country Nono, an enemy of America"

Enemy(Nono, America)

Datalog knowledge base → sets of first-order definite clauses with no function symbols.

Algorithm:

function POL-FC-ASK(KB, α) returns a substitution (or false)

inputs: KB, the knowledge base, a set of first order definite clauses

α, the query, an atomic sentence.

local variables: new, new sentences inferred on each iteration.

repeat until new is empty.

new ← { }

for each sentence r in KB do

$(P_1 \wedge \cdots \wedge P_n \Rightarrow q) \leftarrow$ STANDARDIZE-APART(r)

for each θ such that SUBST(θ, $P_1 \wedge \cdots \wedge P_n$) = SUBST(θ, $P_1' \wedge \cdots \wedge P_n'$)

for some $P_1' \cdots P_n'$ in KB

$q' \leftarrow$ SUBST(θ, q)

if q' is not a renaming of some sentences already in KB or new then do :

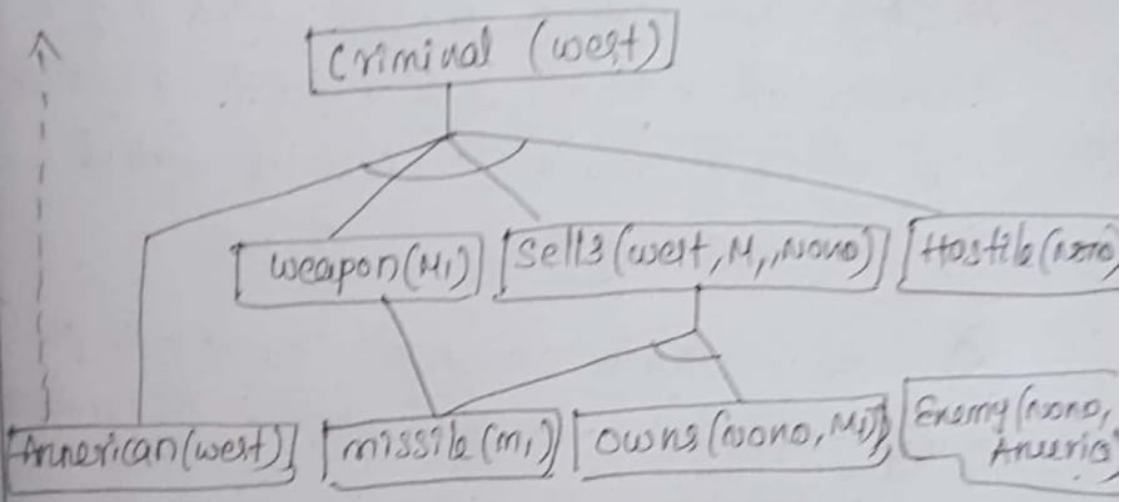
add q' to new

$\phi \rightarrow$ UNIFY(q', α)

if ϕ is not fail then return ϕ

add new to KB

return false.



So, in second iteration the first rule is satisfied with {x/west, y/m1, z/Nono} and criminal (west) is added.

→ It is sound and complete for first-order definite clauses.

→ It will terminate for datalog in finite number of iterations.

→ It may not terminate when α is not entailed. i.e, for example:

Nat Num(0)

$\forall n$ NatNum(n) → NatNum(S(n))

then forward chaining adds

→ Natnum(S(0)), Natnum(S(S(0))), Natnum(S(S(S

and so on ----

This problem is unavoidable in general.

Efficient forward chaining:

(1) the inner loop of algorithm involves finding all possible unifiers such that the Premise of a rule unifies with a suitable set of facts in KB. This is called "Pattern matching"

(2) The Algorithm rechecks every rule on every iteration. to see whether its premises are satisfied / not.

(3) Finally it generate many facts that are irrelevant to goal.

(1) Matching Rules against known facts:

Example:

Missile(x) ⇒ weapon(x)

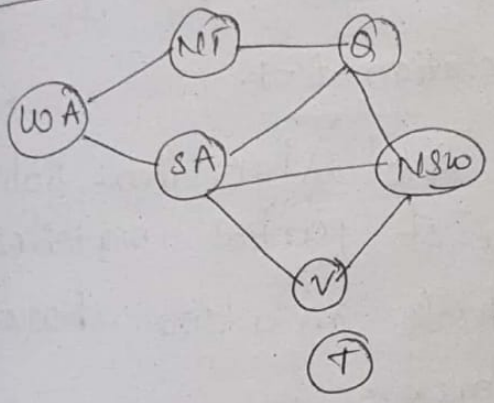
Now we need to find all facts that unify with missile(x) i.e,

missile(x) ∧ owns(Nano, x) ⇒ sells(west, x, Nono)

So, the better way to find this query is first find all missiles and then check whether they owned by Nono. This is called "Conjunct ordering problem".

→ Forward chaining matching Conjuctive Premises against known facts is NP-hard.

forEX: Consider hard matching example.



Diff(wa, nt) ∧ Diff(wa, sa) ∧ Diff(NT, Q) ∧ Diff(nt, sa) ∧ Diff(q, nsw) ∧ Diff(sa, sa) ∧ Diff(nsw, v) ∧ Diff(nsw, sa) ∧ Diff(v, sa) ⇒ colorable

Diff(R, B) Diff(R, G)
Diff(G, R) Diff(G, B)
Diff(Blue, R) Diff(B, G)

B - Blue
G - Green
R - Red

Colorable() is inferred iff CSP has a solution
→ CSP's include 3SAT as a special case,
hence matching is NP-hard.

(2) Incremental forward chaining:

forEx: The rule

Missile(x) ⇒ Weapon(x)

matches against Missile(M_1), and the Conclusion will be Weapon(M_1) is already known so nothing happens.

Such redundant rule matching can be avoided if we make the following observation:

"Every new fact inferred on iteration 't' must be derived from at least one new fact inferred on iteration t-1.

→ Here, the "Rete" Algorithm solves the problem of partial matches. If two literals in a rule share a variable ... forex:

$$Sells(x, y, z) \wedge Hostile(z)$$

then bindings from each literal are folded through an equality node.

(3) Irrelevant facts:

FOL-FC-ASK will generate many irrelevant conclusions.

(1) To avoid these conclusions is to use backward chaining.

(2) Another one is restrict forward chaining to a selected subset of rules.

③ Emerging in deductive database Community, [rewrite ruleset]

i.e, relevant variables can bond this belongs to "magicset".

Backward chaining:

① These algorithms work backward from the goal, chaining through rules to find known facts that support the proof.

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 already applied)
θ, the current substitution, initially empty{}

Localvariables: answers, a set of substitutions, initially empty

if goals is empty then return {θ}

$q' \leftarrow$ SUBST(θ, FIRST(goals))

for each sentence r in KB where STANDARDIZE-APART(r) = $(p_1 \wedge \cdots \wedge p_n \Rightarrow q)$

and $\theta' \leftarrow$ UNIFY(q, q') succeeds

$new\_goals \leftarrow [p_1, p_2, \ldots, p_n | Rest(goals)]$

$answers \leftarrow$ FOL-BC-ASK(KB, new_goals, compose(θ', θ)) ∪ answers

return answers.

→ FOL-BC-ASK is called with a list of goals containing a single element, i.e, the original query. and returns a set of all substitutions satisfying the query.

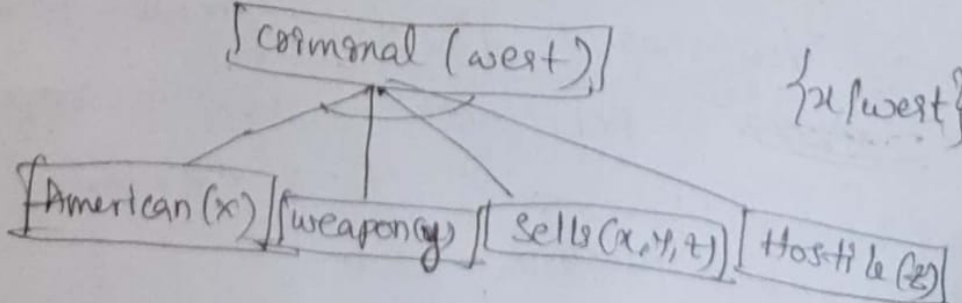
→ List of goals can be considered as "stack" waiting to worked on ; if all of them are satisfied, then the current branch of proof succeeds.

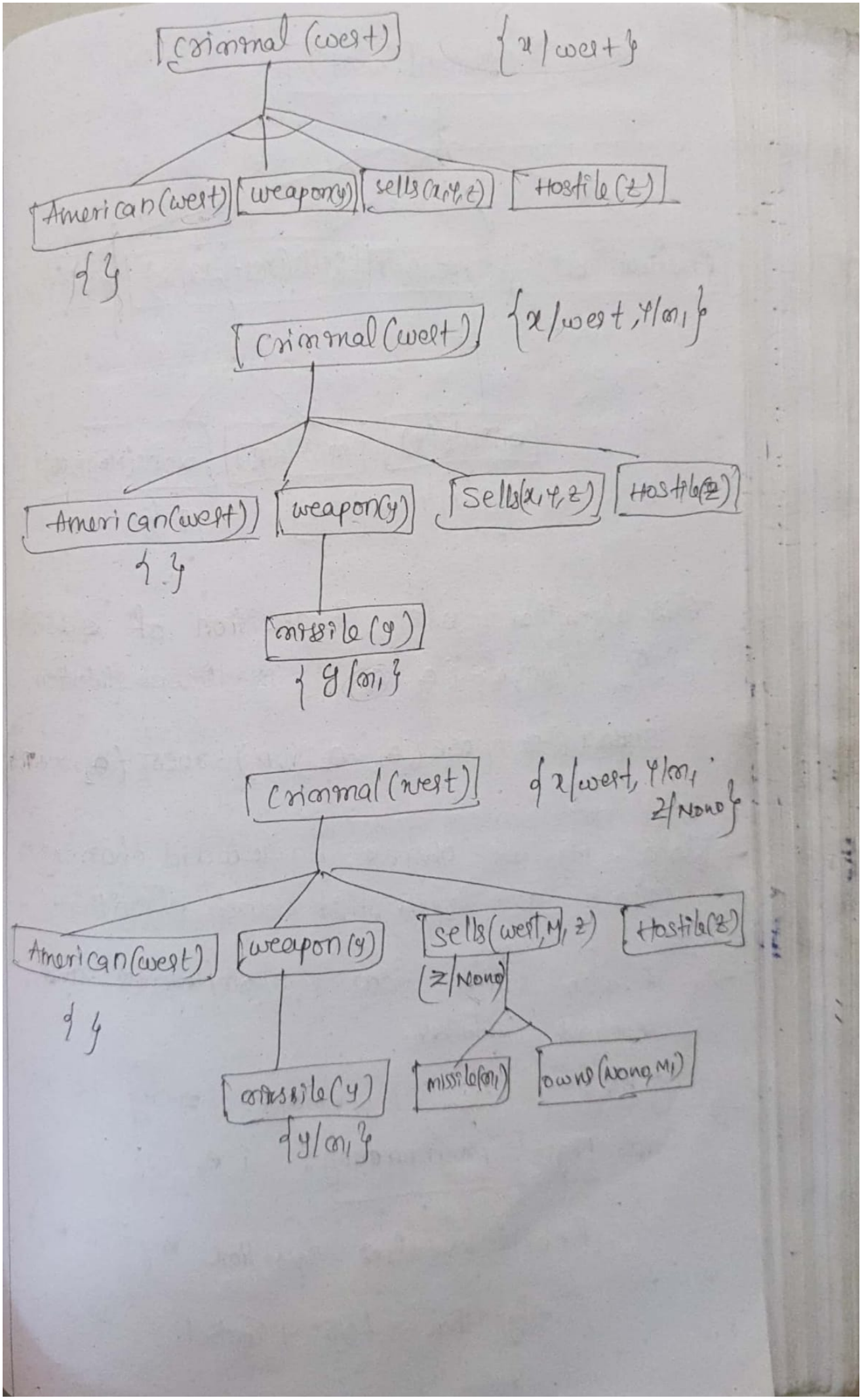
→ The algorithm ~~finds~~ takes first goal in list and find every clause in KB, whose positive literal (or) head unifies with goal.

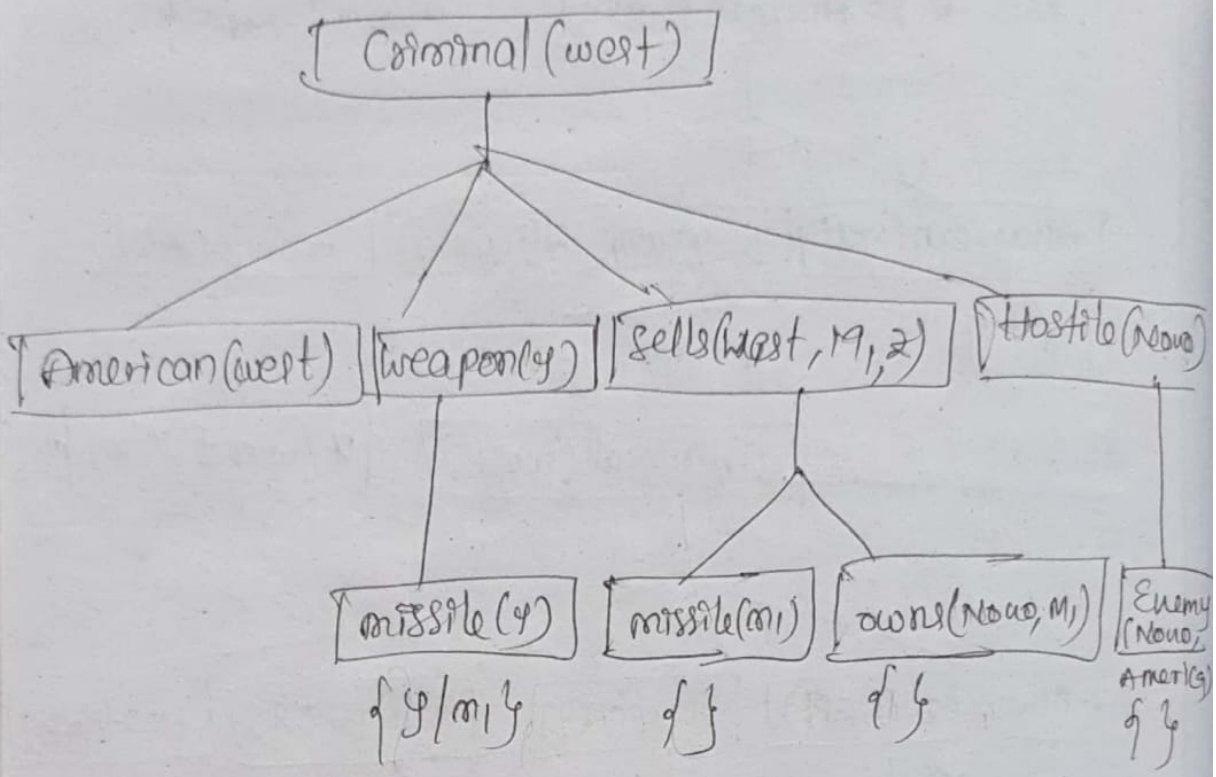
→ Each such clause creates a new recursive call in which the premise (or) body is added to goal stack.

→ If goal unifies with known fact, then no new subgoals are added to stack and goal is solved.

Consider, [Criminal (west)]







This algorithm uses Composition of substitution i.e, Compose(θ_1, θ_2) is the substitution,

SUBST(compose(θ_1, θ_2), P) = SUBST(θ_2, SUBST(θ_1, P))

→ Now. If we observe, backward chaining is clearly a depth first search algorithm.

→ It also suffers from incompleteness and repeated states.

→ Consider how backward chaining is used in Logic programming: i.e,

Robert kowalski equation says that,

Algorithm = Logic + Control.

"PROLOG" is most widely used Logic Programming Language.

Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order-logic.

→ It uses uppercase letters for variables and lowercase letters for constants. clause can be preceding with ":-"

criminal(x) :- american(x), weapon(y), sells (x,y,z), hostile(z).

→ Prolog uses syntactic sugar for list notation and arithmetic

Ex: . append (x, y, z) succeeds if list z is result of appending lists x & y.

append ([], Y, Y)

append ([A|X], Y, [A|Z]) :- append(X, Y, Z)

forexample:

query → append (A, B, [1,2])

solutions will be; A = [] A = [1] A = [1,2]
B = [1,2] B = [2] B = []

→ Execution of prolog programs done via depth-first-backward chaining.

Prolog allows a form of negation called "negation as failure" i.e,

Example: alive(x) :- not dead(x)

read as → "Everyone is alive if not provably dead"

→ prolog has an equality operator '=' but it lacks the full power of logical equality.

i.e, equality of goal succeeds, if two terms are unifiable and fails otherwise.

x+y=2+3 succeeds with X, Y bound to 2, 3

→ the occur check is omitted from prolog unification algorithm.

Efficient implementation of Logic programming:

Execution of prolog program can happen in 2 modes < interpreted / Compiled

→ interpretation runs the FOL-BC-ASK algorithm with program as KB. these interpreters are designed essentially to maximize speed.

① first instead of constructing list of all possible answers for each subgoal

before Continuing to next. These interpreters generate one answer and a "promise" to generate the rest when the answer is fully explored. This promise is called "Choice point".

It also provides a very simple interface for debugging because at all times there is only a single solution path.

(2) FOL-BC-ASK spends a good deal of time in generating & composing substitutions when a path in search fails, prolog will back up to previous choice point, and unbind some variables. this is called "trail" so, the new variable is bound by UNIFY and it is pushed on to trail.

→ Prolog Compilers compile into an intermediate language i.e, warren Abstract Machine (or) WAM, named after David. H. D. Warren. he is one of first implementors of first prolog Compiler. So, WAm is an abstract instructionset that is suitable for prolog and can be either translated / interpreted into machine language.

Continuations: used to implement choicepoints it is packaging up a procedure and a list of arguments that together define what should be done next whenever the current goal succeeds.

→ Here, parallelization also provide substantial speedup. there are 2 types;

1. OR-parallelism
2. AND-parallelism.

(1) Comes from possibility of goal unifying with many different clauses in KB. Each gives an independent search i.e, all branches can be solved in parallel.

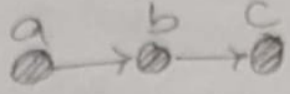
(2) Comes from possibility of solving each Conjunct in the body of an implication in Parallel. it is more difficult because, "Conjunction require Consistent bindings for all variables".

Redundant inference and infinite loops:

Suppose, Consider the logic program that decides if a path exists between two points on a directed graph.

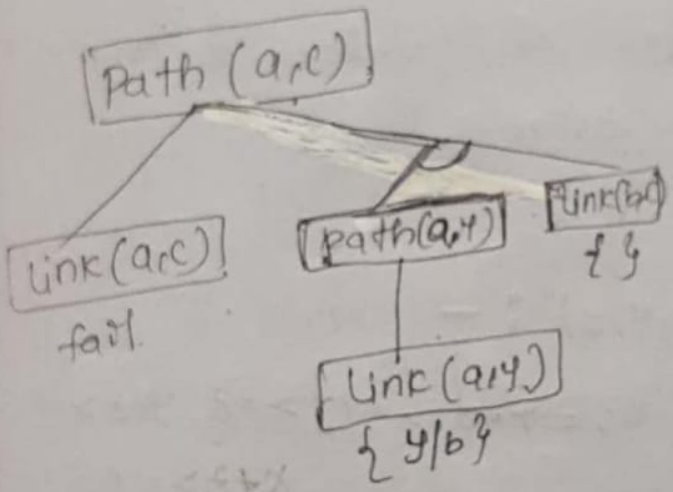
```
Path(x, z) :- link(x, z)
Path(x, z) :- path(x, y), link(y, z).
```

Consider 3 node graph Contains facts link(a,b) and link(b,c)

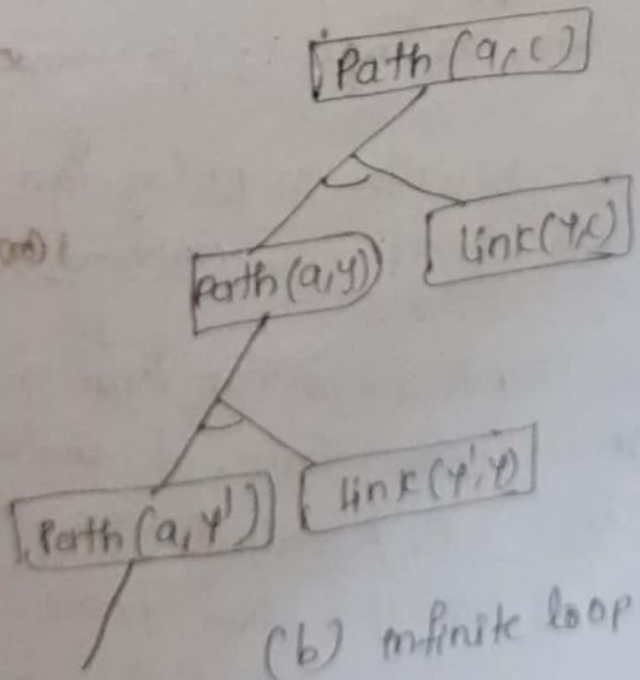


it generates the query path(a,c)

Here, each node is connected to two random successors in the next Layer.



(a) Path exist from A to C



(b) infinite loop when clauses in wrong order

Constraint Logic programming:

The Constraint satisfaction problems (csp's) can be solved in Prolog as same like backtracking algorithm.

Because it works only for finite domain csp's. In prolog terms, there must be finite number of solutions for any goal with unbound variables.

forex:

```
triangle(x,y,z) :-
    x>=0, y>=0, z>=0, x+y>=z, y+z>=x,
    x+z>=y.
```

If we have a query, triangle(3,4,5) works fine. But the query like, triangle(3,4,z) no solution

→ The difficulty is variables in prolog can be in one of two states i.e, unbound (or) bound

Binding a variable to a particular term can be viewed as an extreme form of Constraint namely 'equality'. CLP allows variables to be constrained rather than bound.

The solution to triangle(3,4,z) is Constraint $z>=1$.

Resolution

Here, the Complete proof of procedures Can be directly Concern to mathematics. we need to know Completeness theorem for first-order-logic, showing that any entailed sentence has a finite proof.

So, Apply resolution steps to CNF(KB ∧ ¬α) for Complete FOL.

Conversion to CNF: [Conjunctive Normal form for FOL)

It is a Conjunction of clauses, where each clause is a disjunction of literals.

for ex: "Everyone who loves all animals is loved by someone". $\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x,y)] \Rightarrow [\exists y \text{ Loves}(y,x)]$

① Eliminate bidirections and implications

i.e,

$\forall x . [\neg \forall y \neg \text{Animal}(y) \vee \text{Loves}(x,y)] \vee [\exists y \text{ Loves}(y,x)]$

② Move ¬ inwards:

i.e, $\neg \forall x\ P$ becomes $\exists x \neg P$

$\neg \exists x P$ becomes $\forall x \neg P$

The above sentence Can be transformed as;

$\forall x [\exists y \neg (\neg \text{Animal}(y) \vee \text{Loves}(x,y))] \vee [\exists y \text{ Loves}(y,x)]$

$\forall x [\exists y \neg \neg \text{Animal}(y) \wedge \neg \text{Loves}(x,y)] \vee [\exists y \text{ Loves}(y,x)]$

$\forall x [\exists y \text{ Animal}(y) \wedge \neg \text{Loves}(x,y)] \vee [\exists y \text{ Loves}(y,x)]$

Now, see how a universal quantifier ($\forall y$) becomes an existential quantifier ($\exists y$) with meanin

(3) Standardize variables: Each quantifier should use a different variable name i.e,

$\forall x [\exists y \text{ Animal}(y) \wedge \neg \text{Loves}(x,y)] \vee [\exists z \text{ Loves}(z,x)]$

(4) Skolemize: It is a process of removing existential quantifiers by elimination.

i.e, $\forall x [\text{Animal}[A] \wedge \neg \text{Loves}(x,A)] \vee \text{Loves}(B,x)$

It is entirely different statement i.e, everyone either fails to love a particular animal A (or) is loved by some particular entity B.

But original sentence is allowing each person to fail to love a different animal (or) to be loved by different person.

so, we want skolem entities to depend on x.

$\forall x [\text{Animal}(F(x)) \wedge \neg \text{Loves}(x, F(x))] \vee \text{Loves}(G(x), x)$

F, G are Skolem functions.

To satisfy this sentence with equality of original sentence.

i.e,

(5) Drop universal quantifiers:

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

(6) Distribute $\wedge$ over $\vee$

$[Animal(F(x)) \vee Loves(G(x), x)] \wedge$

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

Now this sentence is an CNF Consists of two clauses.

It is quite unreadable.

The Resolution inference rule:

It is a lifted version of PL resolution rule. In PL literals are Complementary i.e, one is negation of other.

→ In FOL literals are Complementary i.e, one unifies with negation of other.

$$\frac{l_1 \vee \cdots \vee l_k, \quad m_1 \vee \cdots \vee m_n}{SUBST(\theta, l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where UNIFY $(d_i, \neg m_j) = \theta$.

So, we can resolve the two clauses;

$[Animal(F(x)) \vee Loves(G(x), x)]$ and

$[\neg Loves(u, v) \vee \neg Kills(u, v)]$

By eliminating Complementary literals $Loves(G(x), x)$ and $\neg Loves(u, v)$ with unifier, $\theta = \{u/G(x), v/x\}$ to produce resolvant clause.

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

Here it is resolving exactly two literals.

So, this can be called as "Binary Resolution"

But, binary resolution rule by itself doesn't yield a Complete inference procedure.

So, an alternative approach is "factoring".

→ i.e, Removal of Redundant literals to one if they are identical in propositional logic

→ But, first-order factoring reduces two literals to one if they are unifiable.

So, with the combination of {binary resolution + factoring} ⇓ Complete

$\neg m_j) = \phi$.

the two clauses;

$G(x), z)]$ and

loves $(u, v) \vee \neg$ kills $(u, v)]$

tory literals loves $(G(x), x)$

with unifier,

to produce resolvent

$(G(x), x)]$

ty two literals.

as "Binary Resolution"

by itself doesn't

e procedure.

h is "Factoring".

lant literals to one

propositional logic

reduces two literals

sfiable.

of { Binary resolution
Factoring
Complete

Proof : Example:

Resolution proves that $KB \models \alpha$ by proving $KB \wedge \neg\alpha$ is unsatisfiable i.e by deriving empty clause.

suppose Consider the sentences in CNF are;

$\neg$American $(x) \vee \neg$weapon $(y) \vee \neg$sells $(x, y, z) \vee \neg$Hostile $(z) \vee$ criminal (x),

$\neg$ missile $(x) \vee \neg$owns (nono, x) $\vee$ sells (west, x, nono),

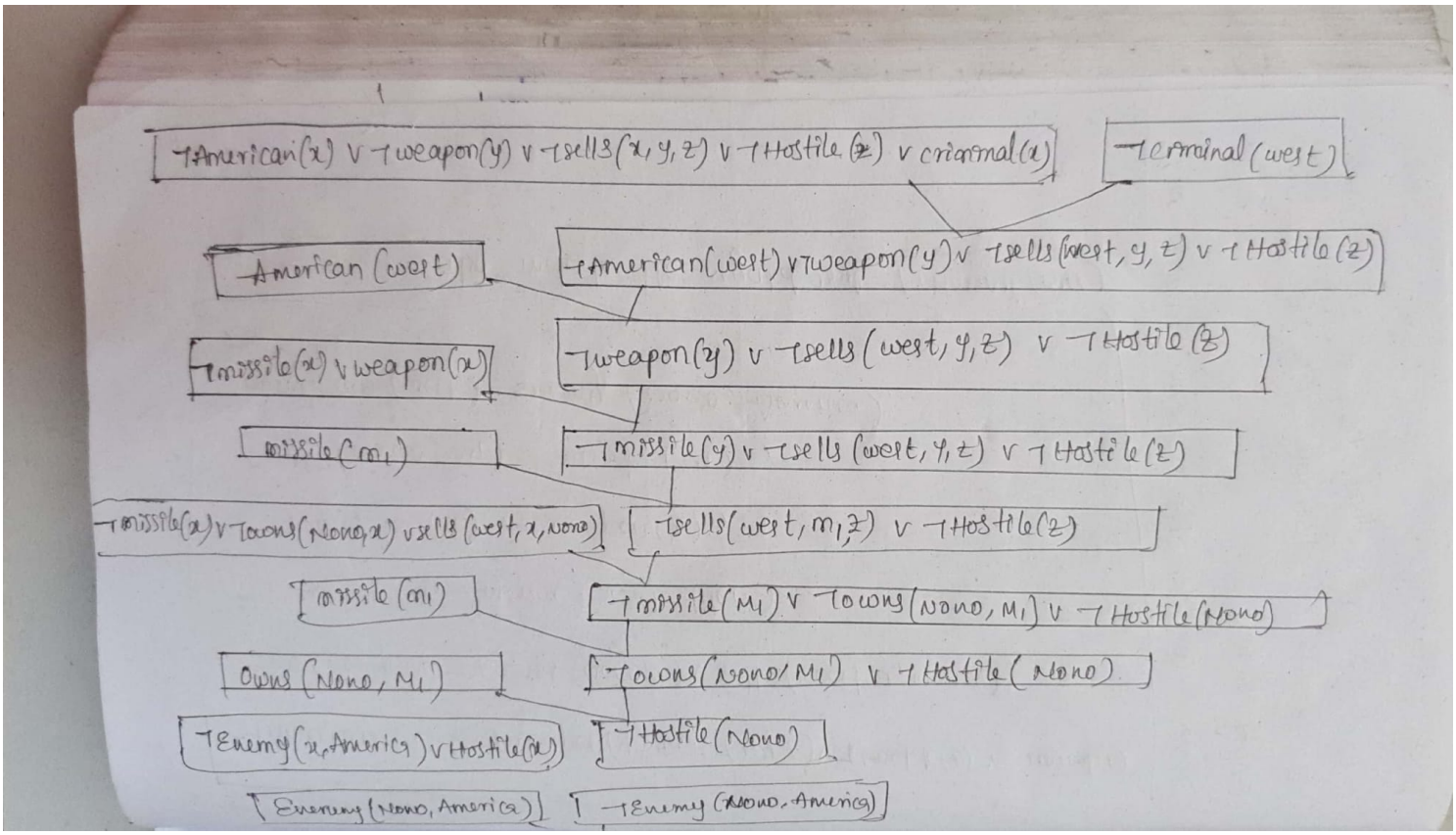
$\neg$ enemy (x, America) $\vee$ Hostile (x),

$\neg$ missile (x) $\vee$ weapon (x),

owns (nono, M_1), missile (M_1),

American (west), Enemy (nono, America).

also include the negated goal $\neg$ criminal (west)



Completeness of Resolution:

Here the resolution is refutation-complete i.e, if a set of sentences is unsatisfiable, then resolution will always be to derive Contradiction. [i.e, negated goal method]

→ A given sentence in FOL can be Rewritten as a set of clauses in CNF.

So, it can be proved by induction on the form of the sentence, using atomic Sentences as base. i.e;

goal is to prove that;

if S is an unsatisfiable set of clauses, then the application of a finite number of resolution steps to S will yield a contradiction

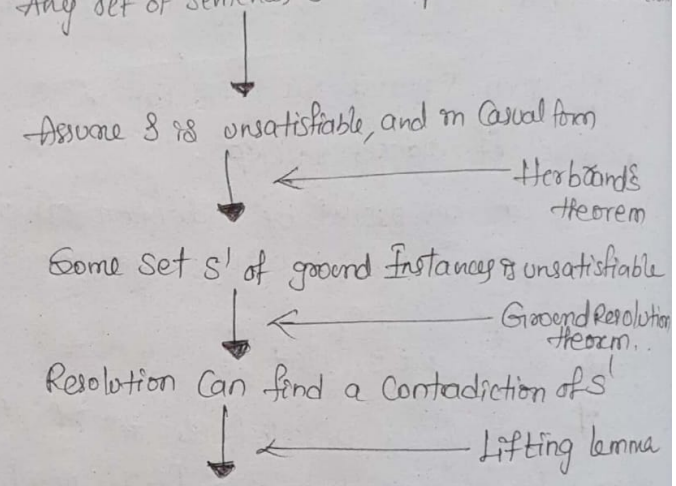
So, the proof will be as follows:

1) first we observe that if S is unsatisfiable, then there exists a particular set of ground instances (or ground instances of clauses of S such that this set is also unsatisfiable.

2) Then, we apply "ground Resolution theorem". i.e, propositional resolution is Complete for ground Sentences.

3) Use "Lifting Lemma" to show that, for any Propositional resolution proof using the set of

ground sentences, there is a corresponding first-order resolution proof using first order sentences from which the ground instances were obtained.



→ To carry first step we need 3 new concepts:

① Herbrand universe: if S is set of clauses, then Hs, the Herbrand universe of S, is set of all ground terms i.e, from
function symbols in S, if any
Constant symbols in S, if any
if none, then it will be constant 'A'

for ex: S contains just the clause
$\neg P(x, F(x,A)) \vee \neg Q(x,A) \vee R(x,B)$.

then, Hs is

{ A, B, F(A,A), F(A,B), F(B,A), F(B,B),
F(A, F(A,A)), ... }

(2) Saturation: if S is a set of clauses & P is set of ground terms, then P(S), the saturation of S with respect to 'P', is set of all ground terms/clauses obtained by applying all possible consistent situations of ~~set S~~ ground terms in 'P' with variables in 'S'.

(3) Herbrand base: The saturation of set 'S' of clauses with respect to its Herbrand universe called Herbrand base of S, written as $H_S(S)$ is infinite set of clauses.

{ ¬P(A, F(A,A)) ∨ ¬Q(A,A) ∨ R(A,B),
¬P(B, F(B,A)) ∨ ¬Q(B,A) ∨ R(B,B),
¬P(F(A,A), F(F(A,A),A)) ∨ ¬Q(F(A,A),A) ∨ R(F(A,A), B),
¬P(F(A,B), F(F(A,B),A)) ∨ ¬Q(F(A,B),A) ∨ R(F(A,B),B), ... }

Herbrand's theorem:

if a set S of clauses is unsatisfiable, then there exists a finite subset of $H_S(S)$ that is also unsatisfiable.

Let S' be finite subset of ground instances

Now we can appeal to ground resolution theorem to show that resolution closure $RC(S')$ contains empty clause.

→ Will start by single application of resolution rule. Robinson's basic lemma implies the following fact.

Let C_1 and C_2 are two clauses with no shared variables, and let C_1' and C_2' be ground instances of C_1 and C_2. If C' is a resolvent of C_1' and C_2', then there exists a clause C such that (1) C is a resolvent of C_1 and C_2 and (2) C' is ground instance of C

Lifting lemma: i.e, It lifts a proof step from ground clauses up to general first-order clauses. To prove lifting lemma Robinson had invented unification and derive all of the properties of most general unifiers Here we simply illustrate Lemma:

i.e,

$C_1 = \neg P(x, F(x,A)) \vee \neg Q(x,A) \vee R(x,B)$

$C_2 = \neg N(G(y), z) \vee P(H(y), z)$

$C_1' = \neg P(H(B), F(H(B),A)) \vee \neg Q(H(B),A) \vee R(H(B),B)$.

$Q' = \neg N(G(B), F(H(B),A)) \vee P(H(B), F(H(B),A))$

$C' = \neg N(G(B), F(H(B),A)) \vee \neg Q(H(B),A) \vee R(H(B),B)$

$C = \neg N(G(y), F(H(y),A)) \vee \neg Q(H(y),A) \vee R(H(y),B)$.

So lifting lemma is easy to derive a similar statements about sequence of applications of resolution rule.

[i.e, for any clause c' in resolution closure S' there is a clause c in the resolution closure of S, such that c' is a ground instance of c and derivation of c is same length as derivation of c'.]

Dealing with equality:

There are 3 distinct approaches

① Axiomatize equality i.e, we can substitute equals for equals in any predicate/function. So we need 3 axioms, one for each function.

$\forall x \quad x=x$

$\forall x,y \quad x=y \Rightarrow y=x$

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

$\forall x,y \quad x=y \Rightarrow (P_1(x) \Leftrightarrow P_1(y))$

$\forall x,y \quad x=y \Rightarrow (P_2(x) \Leftrightarrow P_2(y))$

$$\forall w,x,y,z \quad w=y \wedge x=z \Rightarrow (F_1(w,x)=F_1(y,z))$$
$$\forall w,x,y,z \quad w=y \wedge x=z \Rightarrow (F_2(w,x)=F_2(y,z))$$

(2) demodulation:

It takes a unit clause $x=y$ and substitute y for any term that unifies with x in some other clause.

i.e, For any terms x, y, z ... where UNIFY$(x,z)=\theta$ and $m_n[z]$ is a literal containing z:

$$\frac{x=y, \quad m_1 \vee \cdots \vee m_n(z)}{m_1 \vee \cdots \vee m_n[\text{SUBST}(\theta,y)]}$$

Demodulation is used for simplifying expressions i.e, $x+0=x$, $x'=x$ and soon.

(2) paramodulation:

for any terms x, y, and z, where UNIFY$(x,z)=\theta$

$$\frac{l_1 \vee \cdots \vee l_k \vee x=y, \quad m_1 \vee \cdots \vee m_n[z]}{\text{SUBST}(\theta, l_1 \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_n[y])}$$

(3) Equational unification:

Reasoning entirely within an extended unification algorithm. terms are unifiable if they provably equal under some substitution
↳ equality Reasoning.

1+2 = 2+1 normally not unifiable, but unification Algorithm knows that [x+y = y+x] could unify them with empty substitution

Resolution strategies:

Here we examine strategies that help to find proofs efficiently.

Unit preference:

The idea is to produce an empty clause i.e, one sentence is single literal / unit clause

i.e, sentence P with other sentence (¬P ∨ ¬Q ∨ R) then derives

(¬Q ∨ R).

→ Unit resolution is a restricted one in which every step must involve a unit clause. it resembles forward chaining and it is incomplete, but may be complete with Horn knowledge bases.

Set of support:

It starts by identifying a subset of sentences called set of support. Every resolution combines a sentence from set of support with another sentence and adds the resolvent to set of support. it is Complete one.

input resolution :

Every resolution combines one of the input sentences from KB with some other sentence -- as like [resolution proof in 298 page]

linear resolution : this strategy is allowing P and Q to be resolved together either if P is in the original KB (or) if P is an ancestor of Q in proof tree. it is complete.

⇒ SUBSUMPTION :

It is a method that eliminate all sentences that are subsumed by an existing sentences in KB.

for ex : if P(x) is in KB then, there is no sense in adding P(A).

Theorem provess :

These are differ from Logic programming languages in two ways.

1) most logic programming languages handle only Horn clauses, where as theorem provess accept full first-order-logic.

2) prolog programs intervene logic & control in logic programming instead of using A :- B, C if we use A :- C, B it affects the execution of program.

But, in most theorem provers it may doesn't affects the result.

Design a theorem prover :

Here, we consider OTTER theorem prover in preparing a problem for OTTER the user must divide the knowledge into 4 parts:

→ A set of clauses known as set of support (sos) which defines important facts about the problem.

→ A set of usable axioms that are outside the set of support. which provides background knowledge about problem area.

→ A set of equations known as rewrites or demodulators. Applied in left-right direction for ex: $x+0=x$.

→ A set of parameters and clauses that defines control strategy.. i.e, user specifies a heuristic function to control the search and a filtering function to eliminate some subgoals which are uninteresting.

The Algorithm / theorem prover OTTER moves lightest clause in the set of support to the usable list and adds to usable list some immediate consequences of resolving the lightest clause. It halts when no more clauses in sos

Procedure OTTER (sos, usable)

inputs : sos, a set of support -
usable, background knowledge i.e,
relevant to problem.

repeat
clause ← the lightest member of sos
move clause from sos to usable
PROCESS (INFER(clause, usable), sos)
until sos = [] or a refutation has been found.

---

function INFER (clause, usable) returns clauses
resolve clause with each member of usable
return the resulting clauses after applying
FILTER

Procedure PROCESS (clauses, sos)
for each clause in clauses do
clause ← SIMPLIFY (clause)
merge identical literals
discard clause if it is a tautology
sos ← [clause - sos]
if clause has no literals then a refutation has
been found
if clause has one literal then look for unit
refutation.

Extending prolog: [PTTP: prolog Technology Theorem prover]

To get sound and completeness to prolog[PTTP]. there are 5 significant changes;

→ the occurs check is put back into the unification routine to make it sound.

→ the d-f-s search is replaced by an IDS

→ Negated literals i.e, $\neg P(x)$ are allowed.

→ A clause with n atoms is stored as n different rules. forex:

$A \Leftarrow B \wedge C$ stored as $\neg B \Leftarrow C \wedge \neg A$ and $\neg C \Leftarrow B \wedge \neg A$

This technique is called Locking.

→ Inference is Complete by the addition of linear i/p resolution rule: i.e, resoluing by Contradiction. i.e, $\neg P \Rightarrow P$.

~~user has~~ the main drawback of PTTP is Each inference rule is used by system both in it original form and Contrapositive form

i.e, $(f(x,y) = f(a,b)) \Leftarrow (x=a) \wedge (y=b)$

BUT, PTTP also generate negation.

i.e, $(x \neq a) \Leftarrow (f(x,y) \neq f(a,b)) \wedge (y=b)$.