A Case Study on First Order Theorem Proving

DISSERTATION

Submitted in partial fulfillment of the requirements of the M. Tech. Software Systems Degree programme

By

Ashwani Kumar Dwivedi
2021HS11501

Under the supervision of
Mr. Ganesh Kumar
Lead Engineer, Samsung Research Institute Noida

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN)
(December, 2023)

DISSERTATION SSSIZG629T

A Case Study on First Order Theorem Proving

Submitted in partial fulfillment of the requirements of the M. Tech. Software Systems Degree programme

By

Ashwani Kumar Dwivedi
2021HS11501

Under the supervision of
Mr. Ganesh Kumar
Lead Engineer, Samsung Research Institute Noida

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN)
(December, 2023)

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI

CERTIFICATE

This is to certify that the Dissertation entitled ***A Case Study on First Order Theorem Proving*** and submitted by Mr. ***Ashwani Kumar Dwivedi***. IDNo ***2021HS11501*** in partial fulfillment of the requirements of **DISSERTATION SSSIZG629T**, embodies the work done by him under my supervision.

Signature of the Supervisor

Place:
Date:

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

DISSERTATION SSSIZG629T

Dissertation Title:	*A Case Study on First Order Theorem Proving*
Name of Supervisor:	*Ganesh Kumar*
Name of Student:	*Ashwani Kumar Dwivedi*
ID No. of Student:	*2021HS11501*

Abstract

Theorem proving is an important field of *Computer science* which deals with proving new Theorems based on the given *axioms.* Computer based Automated Theorem Provers have been used in many applications including validating new theorems in **Mathematics** and storing and inferencing data from knowledge base in **Artificial Intelligence** which is used by Logical Agents.

One good practical example of these systems can be an *Automatic Kiosk System developed in Hospitals* which can detect the disease of any patients nased on the symptoms given by him.

This case study Aims on learning how these ATP work internally. More specifically, we are going to learn the working of First order Theorem provers.

First order logic is widely used in computer applications involving Artificial Intelligence to store and infer data in Knowledge based agents. However, the Problem of inference in FOL is **NP hard** and there are multiple ways exist to do this.

Section 1, introduces and explains *Logical Agents* and *Theorem proving.* Along with that, it Introduces us with **Propositional Logic** which will be the base of the First order logic. Sub section 1.3 and 1.4 gives the inference systems in propositional logic and Section 1.5 gives an Introduction of First order logic.

Section 2, explains the **Algorithms** that are used for Inference in First order logic which includes preprocessing steps described in 2.1. in 2.2 - 2.5 explains rule of **Modus ponens** and **Horn clauses**. section 2.6, 2.7 and 2.8 gives three different approaches for designing algorithms for inference.

Section 3, gives a brief on **Completeness and Decidability** in First order logic, which is an important part as it is necessary for any user querying anything in ATPs using FOL must form queries in such a way that the system

should stop at some point. Since, The system has been proven to be a Turing machine.

Section 4, gives a small brief on two ATPs, which are used in industrial programming, and explains the core method used by them. *Section 5*, gives a small brief on the **Software design** of the system that should be implemented in order to make an ATP based on First order logic.

Section 6, gives an introduction on the modern data that has been widely used by the developed systems to validate their theorem provers as concurrently more and more research is going on and with time these provers are becoming faster and faster.

Contents

1 Introduction

1.1 Logical Agents

Knowledge Base Agents are crucial part of Artificial Intelligence. These agents have two functions [Norvig and Intelligence, 2002] TELL and ASK, where TELL operation adds a statement to the knowledge base, and ASK operation may ask the validity of a statement based on the knowledge present with the agent.

Logical Agents are the kind of Knowledge based agents in which the knowledge is represented by a set of statements of so called **axioms**, which will then be used to add new statements which can be **entailed** from the present knowledge base. this process is called **inference**. If a statement α can be entailed from a Knowledge base KB then this can be represented as KB entails α or in notation:

$$KB \models \alpha \tag{1}$$

1.2 Automated Theorem proving

Theorem proving is a part of Logical reasoning that deals with proving theorems or statements using computer based on several axioms that has been fed in order to provide context. these theorem provers require specific type of knowledge representation which can be easily used by the computer programs to evaluate the expression. Most commonly used representation are **Propositional Logic** and **First order logic**.

1.3 Propositional Logic

T he most simplest way to represent knowledge base is propositional logic. [Norvig and Intelligence, 2002]The **atomic** sentences consist of propositional symbol. each symbol stands for proposition that can be either true or false. **Complex** sentences are constructed from simpler sentences using parenthesis and Logical connectives.

- $\neg$ (not): represent negation of a proposition.
- $\wedge$ (and): A $\wedge$ B, conjuncts A with B means the proposition will be true only when both A and B are true.
- $\vee$ (or): A $\vee$ B, is disjuncts A with B means the proposition will be true if either of A and B are true.
- $\Rightarrow$ (implies): A $\rightarrow$ B, if A then B, are similar to rules of **if-then**.
- $\Leftarrow$ (biconditional): A $\leftarrow$ B, A if and only if B.

A complex sentence may also be called a **clause** made of atomic literals. and the knowledge base can be represented as an and of all the clauses.

1.3.1 Proof using Modus ponens

Rule of Modus ponens says that for any two statements α and β. if KB contains α and $\alpha \Rightarrow \beta$ then β can be inferred from the knowledge base. or in notation.

$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta} \tag{2}$$

And-Elimination is another useful rule which states that from a conjunction any of conjuncts can be inferred.

$$\frac{\alpha \wedge \beta}{\beta}, \frac{\alpha \wedge \beta}{\alpha} \tag{3}$$

Horn clauses

Horn clauses are special type of clauses which can have at most one positive literal. e.g $A \vee \neg B \vee \neg C$. The reason for this is so that in implications ($\Rightarrow$), there is only one literal present on the right hand side. e.g $A \vee \neg B \vee \neg C$ can also be written as $C \wedge B \Rightarrow A$,
if our Knowledge base can be converted into a set of horn clauses. the resolution [Norvig and Intelligence, 2002] can be performed very fast using **Forward chaining** or **Backward chaining**.

Forward chaining and Backward chaining Algorithm

Forward chaining involves constantly using rule of Modus Ponens to infer new clauses until the queried clause is inferred from the knowledge base. this process involves constantly checking the KB for new clauses and may also produce some non useful clauses.

Backward chaining involves the start from the clause that need to be inferred and using Modus Ponen goes back towards finding the matching clauses in order to fill the spaces. this approach is similar to Depth First algorithm used in searching the solution.

1.4 Proof using Refutation

This Procedure involves proof by contradiction. suppose we have a clause α and we need to check if KB $\models \alpha$.
Proving $KB \models \alpha$ is same as checking $KB \wedge \neg\alpha$ is unsatisfiable.
Above conjuncture can be easily proven using **Proof by contradiction** [Norvig and Intelligence, 2002].

Refutation uses the same method in order to check weather a clause can be inferred from the knowledge base.

1.4.1 Conjunctive Normal Form

The resolution rule only is applied on disjuncts of literals or clauses. hence the knowledge base must be converted into conjunction of clauses. this form is called Conjunctive Normal Form(CNF).
Any knowledge base can be converted into CNF form using Following rules[Norvig and Intelligence, 2002].

- Eliminate $\Leftrightarrow$: replace $A \Leftrightarrow B$ with $A \Rightarrow B \wedge B \Rightarrow A$
- Eliminate $\Rightarrow$: replace $A \Rightarrow B$ with $\neg A \vee B$
- CNF require $\neg$ to appear in only literals so move $\neg$ inwards where ever necessary.
  - $\neg(\neg A) \equiv A$
  - $\neg(A \wedge B) \equiv \neg A \vee \neg B$ (De Morgan)
  - $\neg(A \vee B) \equiv \neg A \wedge \neg B$ (De Morgan)
- Apply distributive law of $\vee$ over $\wedge$ where ever possible to convert expression into CNF.
  - $A \vee (B \wedge C) \equiv (A \vee V) \wedge (A \vee C)$ (distribute $\vee$ over $\wedge$)
  - $A \wedge (B \vee C) \equiv (A \wedge V) \vee (A \wedge C)$ ($\wedge$ over $\vee$)

1.4.2 SAT problem

Model refers to any assignment of literals so that a CNF instance is satisfied. this problem of finding a model of a CNF instance is SAT problem which itself is a **NP Complete** problem. i.e a polynomial algorithm to give the answer is not likely exist.
However, modern SAT solvers are very fast and the increment in computing power has been added to our favor. two common algorithms most commonly used are **DPLL(Davis Putnam Longemann Loveland)** [Baumgartner et al., 2002], and **WalkSAT** [Hoos and Stützle, 2000] Algorithm.

DPLL ALgorithm

DPLL algorithm checks all possible assignment to symbols like any optimization algorithm. but makes it faster by removing bf pure symbols and **unit clauses** in every iteration.
FIND-PURE-SYMBOL returns set of pure symbols with assignment to them. pure symbols are the symbols which have same polarity in all the clauses. these symbols will always have same assignment regardless of their positions and hence are unnecessary.
FIND-UNIT-CLAUSE returns set of clauses which contain only single literals. these clauses can have only single assignment and hence the symbols that are contained must be assigned and removed.

```
Input: clauses, symbols, model
Output: true or false
if every clause in clauses is true in model then
    return true;
end
if some clause in clauses is false in model then
    return false;
end
P, value ⟵ FIND-PURE-SYMBOL(symbols, clauses, model) ;
if P is non null then
    return DPLL(clauses, symbols - P, model ∪ P = value)
end
P, value ⟵ FIND-UNIT-CLAUSE(symbols, clauses, model) ;
if P is non null then
    return DPLL(clauses, symbols - P, model ∪ P = value)
end
P ⟵ first symbol in symbols;
rest ⟵ symbols - P;
return DPLL(clauses, rest, model ∪ P = true);
or DPLL(clauses, rest, model ∪ P = false);
```

Algorithm 1: DPLL Algorithm [Norvig and Intelligence, 2002]

WalkSAT Algorithm

Though DPLL algorithm gives us a complete and sound solution. for larger number of literals in clauses the algorithms is much slower. for this problem Many Optimization algorithms can be used to search for a solution in the solution space, considering the solution space consist of a finite number of models. Some famous algorithms that can be used are Simulated Annealing, Genetic algorithm etc.

One such algorithm is **WalkSAT** , which selects a random path at every iteration and returns true if it finds an assignment. however sometimes it might take a higher time to converge to the right conclusion, but they are observed to perform faster and better in large datasets given sufficient iterations.

```
Input: clauses, p (probability of choosing a random walk), max-flips
       (number of flips allowed before giving up)
Output: satisfying model or faliure
model ⟵ a random assignment of true/false to symbols in clauses;
foreach i in 1..max-flips do
    if model satisfies clauses then
        return true;
    end
    clause ⟵ a randomly selected clause from clauses that is false in
      model;
    with probability p flip the value of a randomly selected symbol
      from clause in model;
    otherwise flip whichever symbol in clause maximizes the number of
    satisfied clause;
end
```

Algorithm 2: WalkSAT Algorithm [Norvig and Intelligence, 2002]

1.5 First order logic

Though , many real world problems might be expressed in propositional logic, however, it is **not expressive enough** to contain more complex statements like *Ramesh is father of Samarth*.
A more expressive logic is First order logic or **Predicate logic**. which includes *objects and relations* along with the facts, while only facts can be represented by the propositional logic.

1.5.1 Components of First order logic

Ground terms

Ground terms are symbols indicating objects.

Terms

Terms are the logical expression that refers to an object. e.g Ram, Ball, Father(peter), Leg(Ramesh) etc.

Functions

Functions map one ground symbol to another symbol. e.g LeftLeg(Sam) indicates the left leg of Sam. Father(Ram) indicates ram has a father and is represented as Father(Ram). similarly father of father of Ram can be written as Father(Father(Ram)).

Atomic Sentences

An atomic sentence is formed from a predicate symbol optionally followed by a parenthesized list of therms[Norvig and Intelligence, 2002]. e.g Brother(Richard, John), Married(Father(Peter), Mother(Peter)).

Complex Sentences

Complex Sentences are made by combining Atomic sentences with logical connectives $(\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow)$. e.g
$\neg King(Richard) \Rightarrow King(John)$ indicates that if Richard is not a king then john is the king.

Quantifiers

First order logic contain two quantifiers Universal($\forall$) and Existential($\exists$). These help in generalizing relations over symbols.
"For all x, if x is a king then x is a person"
$\rightarrow \forall x, king(x) \Rightarrow person(x)$
"There exist x such that x is a crown and x is present on head of john"
$\rightarrow \exists x, crown(x) \wedge onHead(x, John)$.
in both above statements x is a **variable** and can take any term. Any term with no variable is called **Ground term**.

Equality

Equality can be used to represent assertion of equality between terms. e.g.
$Father(John) = Henry$
above statement asserts that Henry is the father of John.

2 Inference in First order logic

2.1 Removing quantifiers

Universal and Existential quantifiers must be removed in order for our algorithms to work for this we do **Universal Instantiation** and **Existential Instantiation**.

2.1.1 Universal Instantiation(UI)

Rule of UI states that we can infer any sentence obtained by substituting **ground term** for a variable. hence for removing universal instantiation we can just replace the variables with all the ground terms present inside the knowledge base.

2.1.2 Existential Instantiation(EI)

Rule of EI states that the variable in sentence must be replaced by a *single constant symbol.* this symbol is called **Skolem constant**, and the process is called **Skolemization**.

2.2 Generalized Modus Ponens

Rule of generalized Modus ponens state that for atomic sentences p_i, p'_i and q. where there is a substitution θ such that $SUBST(\theta, p'_i) = SUBST(\theta, p_i)$ for all i, then

$$\frac{p'_1, p'_2, ..., p'_n, (p_1 \wedge p_2 \wedge ... \wedge p_n \Rightarrow q)}{SUBST(\theta, q)} \tag{4}$$

2.3 Unification

The core part of inference in First order logic is unification. the $UNIFY$ Algorithm takes two clauses/sentences and return a **unifier** is there exist one.

$$UNIFY(p, q) = \theta, \tag{5}$$

where $SUBST(\theta, p) = SUBST(\theta, q)$

e.g. $UNIFY(knows(A, x), knows(y, B)) = x/B, y/A$

Following Algorithm explains the working of Unification Algorithm.

2.4 Storage and Retrieval

STORE and FETCH functions are used to store and fetch data into and from the knowledge base similar to **TELL and ASK** in propositional logic. **STORE(s)** function stores a new sentence s in the knowledge base and **FETCH(q)** returns all **unifiers** such that query q unifies with some sentences in knowledge base.

```
Input: x(statement),y(statement),$\theta$(current substitution)
Output: Most General Unifier / a substitution to make x and y
        identical
if $\theta$ is fail then
    return fail
end
else if $x$ is variable then
    return $UNIFY - VAR(x, y, \theta)$
end
else if $y$ is variable then
    return $UNIFY - VAR(y, x, \theta)$
end
/* compound sentences have functions;
e.g F(A, B), Op field contains F, and Args field contains A and B.      */
else if $x$ is compound and $y$ is compound then
    return UNIFY(x.Args, y.Args, UNIFY(x.Op, y.Op, $\theta$))
end
else if $x$ is list and $y$ is list then
    return UNIFY(x.Rest, y.Rest, UNIFY(x.First, y.First, $\theta$))
end
else
    return fail
end
```

Algorithm 3: UNIFY Algorithm [Norvig and Intelligence, 2002]

```
Input: var(variable or constant),x(complex term),$\theta$(substitution)
Output: a substitution
/* check if $\theta$ contains a substitution of var                      */
if $var/val \in \theta$ then
    return UNIFY(val, x, $\theta$)
end
/* check if $\theta$ contains a substitution of x                        */
else if $x/val \in \theta$ then
    return UNIFY(var, val, $\theta$)
end
/* check if variable var occurs inside x                                 */
else if $OCCUR - CHECK(var, x)$ then
    return fail
end
else
    return add($\{var/x\}$ to $\theta$)
end
```

Algorithm 4: UNIFY-VAR Algorithm [Norvig and Intelligence, 2002]

2.5 Horn Clause

Similar to Propositional logic, in First order logic a Horn clause is a clause which can have at most one positive term. This allows us to use Forward or Backward chaining for inference.

2.6 Forward Chaining

Similar to propositional logic Forward chaining relies on *Generalized Modus Ponens* rule to keep adding new statements into the knowledge base until the queried statement is inferred or no new statements can be added.

```
Input: KB (Knowledge Base), $\alpha$ (queried statement(atomic sentence))
Output: a substitution or false
new ⟵ {};
while new is not empty do
    new ⟵ {};
    foreach rule in KB do
        $p_1 \wedge p_2 \wedge ... \wedge p_n \Rightarrow q$ ⟵ rule;
        foreach $\theta$ such that
         $SUBST(\theta, p_1 \wedge p_2 \wedge ... \wedge p_n) = SUBST(\theta, p'_1 \wedge p'_2 \wedge ... \wedge p'_n)$ do
            q' ⟵ SUBST($\theta$, q);
            if q' does not unify with some sentence in KB or new then
                add q' to new;
                $\phi$ ⟵ UNIFY(q', $\alpha$);
                if $\phi$ is not fail then
                    return $\phi$
                end
            end
        end
    end
    add new to KB;
end
```
Algorithm 5: FOL-FC-ASK Algorithm [Norvig and Intelligence, 2002]

2.7 Backward Chaining

Backward chaining starts from the goal query and then move backwards to searching for the substitution in Depth First Fashion.

```
Input: KB (Knowledge Base), $\alpha$ (queried statement(atomic sentence))
Output: a set of substitutions
return FOL-BC-OR(KB, $\alpha$, );
```
Algorithm 6: FOL-BC-ASK Algorithm [Norvig and Intelligence, 2002]

```
Input: KB (Knowledge Base), $\alpha$ (queried statement(atomic sentence),
       $\theta$ (current substitution))
Output: a set of substitutions
$\Theta = \{\}$;
/* $FETCH-RULES-FOR-GOAL(KB, goal)$ returns rules of type
   $lhs \Rightarrow rhs$ from KB such that the rhs unify with goal          */
Rules $\longleftarrow$ FETCH-RULES-FOR-GOAL(KB, $\alpha$);
foreach rule in Rules do
    (lhs, rhs) $\longleftarrow$ rule;
    foreach $\theta_1$ in FOL-AND(KB, lhs, UNIFY(rhs, goal, $\theta$)) do
        $\Theta = \Theta \cup \theta_1$;
    end
end
return $\Theta$;
```
Algorithm 7: FOL-OR generator function [Norvig and Intelligence, 2002]

```
Input: KB (Knowledge Base), $\alpha$ (set of goals), $\theta$ (current
       substitution))
Output: a set of substitutions
$\Theta = \{\}$;
if $\theta = fail$ then
    return fail;
end
else if $\alpha = \{\}$ then
    return $\theta$
end
else
    first $\longleftarrow$ First($\alpha$);
    rest $\longleftarrow$ Rest-Except-First($\alpha$);
    foreach $\theta_1$ in FOL-OR(KB, SUBST($\theta$, first), $\theta$) do
        foreach $\theta_2$ in FOL-AND(KB, rest, $\theta_1$) do
            $\Theta \longleftarrow \Theta \cup \theta_2$;
        end
    end
end
return $\Theta$;
```
Algorithm 8: FOL-AND Algorithm [Norvig and Intelligence, 2002]

Many modern languages like **Prolog** [Colmerauer, 1990], which do apply certain constraints on the Knowledge base so that only Horn clauses are present, use Backward chaining for inference as it is very faster in converging to the solution.

2.8 Proof by Refutation

2.8.1 Conjunctive Normal Form for First Order Logic

Similar to Propositional logic CNF in FOL is the conjunction of disjunction of literals. however, the literals can contain variables which will then be used to find the proper substitution.
Conversion of KB to CNF follows similar procedure like propositional logic.

- Eliminate implications ($A \Rightarrow B \equiv \neg A \vee B$).
- Move $\neg$ inwards where ever necessary.
  - $\neg(\forall xp) \equiv \exists x \neg p$
  - $\neg(\exists xp) \equiv \forall x \neg p$
- Standardize variables: sometimes same variable name is used in complex sentences like $(\exists xP(x)) \vee (\exists xQ(x))$. in this case, one of the variable name needs to be changed.
- Skolemization Though in sentences containing a single quantifiers and ground terms, **Skolem constants**(a completely new entity) can be introduced. However, when dealing with **variables**, if a complex sentence contain quantifier along with the variable, **Skolem Functions** are introduced so that the variables which are not **quantified** will be replaced by a function of quantified variable. e.g [Norvig and Intelligence, 2002]
  - $\forall x[\exists y Animal(y) \wedge \neg Loves(x,y)] \vee [\exists x Loves(z,x)]$
  - $\forall x[Animal(F(x)) \wedge \neg Loves(x,F(x))] \vee [Loves((G(x)),x)]$
- Drop Universal Quantifiers($\forall$): At this point these can simply be dropped from sentences.
- Distribute $\vee$ over $\wedge$.

2.8.2 Resolution Inference Rule

Similar to Propositional logic, two propositional literals are**complementary** if one is negation of the other, in First order logic two propositional literals are complementary if one **UNIFIES** with negation of other.
For two clauses $LandM$.
$L = l_1 \vee l_2 \vee ... \vee l_x$
$M = m_1 \vee m_2 \vee ... \vee m_y$

let, $\theta \longleftarrow UNIFY(l_j, \neg m_j)$

$$\frac{L, M}{SUBST(\theta, l1 \vee ... \vee l_j - 1 \vee l_j + 1 \vee ... \vee l_x \vee m_1 \vee ... \vee m_j - 1 \vee m_j + 1 ... \vee m_y)} \tag{6}$$

The process of resolving two complimentary clauses is called **Binary Resolution**.

2.8.3 Entailment

Similar to Propositional logic in order to prove that $KB \models \alpha$ for some query α and Knowledge base KB. We will need to prove $KB \wedge \neg\alpha$ is unsatisfiable or in language of First Order Logic the unification of KB and $\neg\alpha$ fails.

$$UNIFY(KB, \neg\alpha) = fail \tag{7}$$

Many modern First order Theorem Provers like VAMPIRE [Kovács and Voronkov, 2013] and AVATAR [Voronkov, 2014] use Proof by Refutation for Theorem proving.

3 Completeness and Decidability

3.1 Soundness

A proof system is **sound** if and only if *everything that is provable is actually true given by the system.* i.e if a statement α can actually be entailed from the KB then our proof system will always produce true when asked $KB \models \alpha$.

3.2 Completeness

A proof system is **complete** if and only if *everything that is true given by the system has a proof.* i.e if $KB \models \alpha$ is given by the system then it is guarantee that α can be proven from KB.

3.3 Decidability

A **problem** is said to be decidable if and only if there exist an algorithm that can answer the problem correctly. e.g. in case of inference, the system must be able to tell us correctly if a statement can be entailed from the knowledge base.

3.4 Inference in Propositional Logic

Given a KB entirely in propositional logic. it contains a finite number of symbols, and every symbol can have at most two values(true or false). Thus in order to find if KB entails a statement α we can easily determine that using truth table of $KB \wedge \neg\alpha$. Hence, the Inference in Propositional Logic is **Decidable**.

3.4.1 soundness and completeness

In propositional logic **Forward chaining and Backward chaining** will always produce a finite amount of symbols when data is present in horn clauses. hence, Inference is **sound and complete**, However, when KB is not in **Horn clauses**, then both methods could not be applied.

Proof using Resolution is also **sound and complete** and can be used regardless of the nature of knowledge base. However, Resolution might be computationally expensive than Forward and backward chaining.

3.5 Inference in First Order Logic

In first order logic, we have multiple variables, and each can take any value of a constant symbol or a constant value when using Skolemization. However, when Functions are present in the KB. we can have infinite set of values for variable. e.g. $Father(x), Father(Father(x)), Father(Father(Father(x)))...$

During Inferencing this may lead to any algorithm keep producing symbols indefinetly.

Fortunately, [Abeles, 1994] proves that, *if a sentence is entailed by the original First order database, then there is a proof involving just finite subset of propositionalized Knowledge Base.*

Hence, if originally is a sentence can be entailed, then an algorithm can tell that in finite time. However, if a statement that could not be entailed, the algorithm may never be able to tell.

Alan Turing[1936], Alonzo Church[1936][Chimakonam, 2012] proved that:
*The question of enatilment for first order logic is **Semidecidable**.*
i.e. Algorithms exist that say yes to every entailed sentence, but no algorithm exists that say no to every non entailed sentence.

Note: *When using proof systems the queries must be formulated such that the statement should be entailed. this can be done by smartly writing the query statements so that proving systems can halt at a conclusion.*

3.5.1 Forward chaining and Backward chaining

Both Forward and Backward chaining are **sound and complete** for **Horn clauses**, but cannot be used in general First order logic.

3.5.2 Resolution

Resolution **sound** and is only **Refutation complete**(*not complete*), because, it can only check if a query is entailed by the KB. However, it may not be able to produce new statements or answer variable substitutions in many cases in queries.

Refutation completeness

A proof system is **Refutation complete** if and only if *it can derive false from every unsatisfiable formula.* i.e. in inference system if KB does not entail α then it must give false when asked if $KB \models \alpha$.

Resolution can be simply look upon as a fast mechanism of checking if there is a path to the goal statement. However, finding substitutions can be achieved by **smartly writing the query statements** so that the resolution search can be maximized so that it would be able to find proper substitutions[Norvig and Intelligence, 2002].

4 Modern First order Theorem provers

4.1 Prolog

Prolog is one of the most famous industrial programming language. the statements in prolog are written in form of **Horn clauses**, in this way it preserves completeness and decidability of the solution.
Prolog is loosely based on **Backward Chaining** Algorithm to answer the queries.
here is an example of prolog code:

```
man(rahul).
person(X):- man(X).
male(X):- man(X).
female(X):- \+ man(x).
female(samita).
married(samita, rahul).
loves(X, Y):- married(X, Y).
married1(X, Y):- married(Y, X).
married1(Y, X):- married(X, Y).
child(sita, samita, rahul).
loves(X, Y):- child(X, Y , Z).
loves(X, Z):- child(X, Y , Z).
```

here is an example of query in prolog.

```
?- loves(sita, X).
X = samita.

?- married(samita,rahul).
true.

?- married(samita,sita).
false.
```

4.2 VAMPIRE

VAMPIRE [Kovács and Voronkov, 2013] is a First order theorem prover developed in 2013. the syntax of query writing supported by Vampire is TPTP[Sutcliffe et al., 1994](explained in section 6), which is currently a standard for modern theorem provers.
here are the key advantages of Vampire.

- Vampire is very fast.
- It implements a unique limited resource strategy, which can be used whenever there are constraints on time or computing resources.
- It supports symbol elimination. which, can be used internally for rule generation from programs.

- it can run parallel proofs using multi threading.

Key advantage of using Vampire is it can be used general First order logic for Inference.

4.2.1 Inference System

The proof System used by Vampire is **Proof by Refutation**. as explained earlier this includes process of adding negation of conjencture to the set of axioms and then checking if the set of formulas are unsatisfiable.

Vampire uses ***Superposition Inference System***[Kovács and Voronkov, 2013] which is a method used in the resolution algorithm. This principle is important to implement whenever the Knowledge base contains ***equality***. Explaining the complete formulas here is beyond the scope of this report.

Over the time Vampire had won many competitions in the field of Logic reasoning like ***CASC (The CADE ATP System Competition)***. The same information can checked on official tptp website *https://www.tptp.org/CASC/*.

5 Data set collection

5.1 TPTP

Thousands of Problems for Theorem Provers [Sutcliffe et al., 1994] is a library of test problem for Automated Theorem Proving(ATP)[Bibel, 2013] systems. The TPTP supplies the ATP community with:

- A comprehensive library of the ATP test problems that are available today, in order to provide an overview and a simple, unambiguous reference mechanism.
- A comprehensive list of references and other interesting information for each problem.
- Arbitrary size instances of generic problems (e.g., the N-queens problem). A utility to convert the problems to existing ATP systems' formats.
- General guidelines outlining the requirements for ATP system evaluation. Standards for input and output for ATP systems.

5.1.1 TPTP syntax format

A **problem** is a list of **formulae**. In **THF, TFF, and FOF** the formulae are *typically any number of **axioms** and one **conjecture**.* In **CNF** the formulae are *typically any number of **axioms** and one or more negated **conjectures**.* The top level building blocks of TPTP files are annotated *formulae, include directives, and comments.* An annotated formula has the form:

$$language(name, role, formula, source, usefulinfo) \tag{8}$$

The *source and useful information are optional.* The languages currently supported are **thf** - *typed higher-order form*, **tff** - *typed first-order form*, **fof** - *(full) first-order form*, and **cnf** - *in clause normal form.*
A **problem** is a list of annotated formulae, typically without the source or useful information.

Name: The name is a word starting lower case.

Role: The role is typically one of axiom for axioms, conjecture for conjectures, or negated_conjecture for negated conjectures.

Logical formula: The logical formula uses a consistent and easily understood notation that can be seen in the BNF for THF, BNF for TFF. BNF for FOF, and BNF for CNF.

Example

Here is a toy FOF problem, to prove the conjecture from the axioms (not all the axioms are needed for the proof).

```
%------------------------------------------------------------------------------
%----All (hu)men are created equal. John is a human. John got an F grade.
%----There is someone (a human) who got an A grade. An A grade is not
%----equal to an F grade. Grades are not human. Therefore there is a
%----human other than John.
fof(all_created_equal,axiom,(
! [H1,H2] :
( ( human(H1)
& human(H2) )
=> created_equal(H1,H2) ) )).

fof(john,axiom,(
human(john) )).

fof(john_failed,axiom,(
grade(john) = f )).

fof(someone_got_an_a,axiom,(
? [H] :
( human(H)
& grade(H) = a ) )).

fof(distinct_grades,axiom,(
a != f )).

fof(grades_not_human,axiom,(
! [G] : ~ human(grade(G)) )).

fof(someone_not_john,conjecture,(
? [H] :
( human(H)
& H != john ) )).
%------------------------------------------------------------------------------
```

6 Program Structure

6.0.1 What Proof Method Should be used ?

For using **Forward Chaining or Backward Chaining**, the knowledge base or set of axioms have to be constrained to Horn Clauses. However TPTP, does contain more complex Instances along with the Horn Clauses. Since, we are building this purely for research purposes we will go with **Proof using Resolution** for this time.

6.0.2 Program Structure: Storing of Knowledge base

TPTP language contain various symbols and statements. hence, a parser will be needed for parsing the tokens provided in the program.
For storing the **statements** C++ Set data structure can be used.

```
class KB {
        set<Statement >:: statements;
}
```

Every **statement** itself is a data structure which contain a set of clauses.

```
class Statement {
        set<Clause >:: clauses;
}
```

Every **clause** will have multiple literals.

```
class Clause {
        set<Literal >:: literals;
}
```

Every literal may contain following parts:

- negation symbol.
- a single predicate symbol.
- variables.
- ground symbol.
- Functions.

Each of the above are simply strings which can be mapped using a Hash Map to an uniquely assigned integer. Although every other symbol can be stored, however predicates and Functions are complex terms.
The difference lies in the use of it:

A Predicate can take more than one terms and always return either true or false. However, a function can take more than one terms and returns a new Symbol that has to be fed into the knowledge base.

Now to solve this problem, the concept of ground term must be there. every Ground term must be stored separately in a different Set and its string mapping must be stored.

Hence, we introduce class **Term**.

```
enum TermType {
        Constant,
        Variable
}
class Term {
        enum TermType:: type;
        int:: value;
}
```

Along with that we can have **Functions** and **Predicates** combine together as Actor.

```
enum ActorType {
        Function,
        Predicate
}
class Actor {
        bool:: negation;
        enum ActorType:: type;
        set<Actor>:: actors;
        set<Term>:: terms;
}
```

In above fashion the **recursion** among **predicates and functions** can be established.
Hence, considering this The class **Clause** can be modified as Follows.

```
class Clause {
        set<Actor>:: actors;
}
```

In the above fashion, given no **Quantifiers** present in KB, the statements can be easily stored.

6.0.3 Program Structure: dealing with quantifiers

As described in Section 2. Universal and Existential quantifier must be detected during pre processing the data. within the ground terms Universal and Existential quantifiers can be dealth with **Skolemization** and **Elimination**.

However, because functions are present in the knowledge base, this task will take infinite amount of time, as new symbols keep on adding indefinetly. To resolve this, these special statements with **Universal quantifiers**. must be kept while inference engine is running so that the new symbols are added only when necessary.

6.0.4 Future Planning

Since, this is a partial Project work, we have only introduced the program basics. In our future report we will add the complexity and the complete implemented Software structure along with the code repository.

7 Conclusion and Future planning

In this report we have successfully studied about The First Order Logic. How it is expressed and what challenges will be faced while designing an Inference System for First Order Logic.

In our next report we will create our own Inference System based on the First Order Logic based on programming language C++. The system will have a Command Line Interface for compiling and querying the system. However our long goal is to make a more complex system with a Graphical User Interface to be able to check and debug programs easily.

References

[Abeles, 1994] Abeles, F. (1994). Herbrand's fundamental theorem and the beginning of logic programming.

[Baumgartner et al., 2002] Baumgartner, P. et al. (2002). A first-order logic davis-putnam-logemann-loveland procedure. *AI in the new Millenium, Morgan Kaufmann, Seattle.*

[Bibel, 2013] Bibel, W. (2013). *Automated theorem proving.* Springer Science & Business Media.

[Chimakonam, 2012] Chimakonam, J. O. (2012). *Proof in Alonzo Church's and Alan Turing's Mathematical Logic: Undecidability of First Order Logic.* Universal-Publishers.

[Colmerauer, 1990] Colmerauer, A. (1990). An introduction to prolog iii. *Communications of the ACM*, 33(7):69–90.

[Hoos and Stützle, 2000] Hoos, H. H. and Stützle, T. (2000). Local search algorithms for sat: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481.

[Kovács and Voronkov, 2013] Kovács, L. and Voronkov, A. (2013). First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer.

[Norvig and Intelligence, 2002] Norvig, P. R. and Intelligence, S. A. (2002). A modern approach. *Prentice Hall Upper Saddle River, NJ, USA: Rani, M., Nayak, R., & Vyas, OP (2015). An ontology-based adaptive personalized e-learning system, assisted by software agents on cloud storage. Knowledge-Based Systems*, 90:33–48.

[Sutcliffe et al., 1994] Sutcliffe, G., Suttner, C., and Yemenis, T. (1994). The tptp problem library. In *Automated Deduction—CADE-12: 12th International Conference on Automated Deduction Nancy, France, June 26–July 1, 1994 Proceedings 12*, pages 252–266. Springer.

[Voronkov, 2014] Voronkov, A. (2014). Avatar: The architecture for first-order theorem provers. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 696–710. Springer.