MELİKŞAH
ÜNİVERSİTESİ
KAYSERİ

The Graduate Institute of Science and Engineering

M.Sc. Thesis in Computer Engineering

A Comparative Study of Theorem Provers

by

Adamu Sani YAHAYA

July 2014
Kayseri, Turkey

A Comparative Study of Theorem Provers

By

Adamu Sani YAHAYA

A thesis submitted to

the Graduate Institute of Science and Engineering

of

Meliksah University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

July 2014
Kayseri, Turkey

APPROVAL PAGE

This is to certify that I have read the thesis entitled "A Comparative Study of Theorem Provers" by Adamu Sani Yahaya and that in my opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science in Electrical and Computer Engineering, the Graduate Institute of Science and Engineering, Melikşah University.

July 17, 2014 Prof. Dr. Murat UZAM

Head of Department

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

July 17, 2014 Yrd. Dr. Aytekin VARGÜN

Supervisor

Examining Committee Members

Title and Name		Approved
Asst. Prof. Aytekin VARGÜN	July 17, 2014	
Assoc. Prof. Dr. Ahmet UYAR	July 17, 2014	
Asst. Prof. Hasan PALTA	July 17, 2014	

It is approved that this thesis has been written in compliance with the formatting rules laid down by the Graduate Institute of Science and Engineering.

Prof. Dr. M. Halidun Keleştemur

Director

July2014

A Comparative Study of Theorem Provers

Adamu Sani YAHAYA

M.S. Thesis in Electrical and Computer Engineering
July 2014

Supervisor: Asst. Prof. Aytekin VARGÜN

ABSTRACT

Theorem provers and proof assistants are mainly used for solving mathematical problems or in formal verification of software and hardware. It is critical to deliver correct and safe code to the customers to prevent high cost of software maintenance. This is also indispensable and can cause life-threatening problems in applications that are related to health, aerospace and nuclear reactors etc. Formal verification via theorem provers provides high assurance of correctness and safety. The theorem proving process requires writing specifications and proofs of properties that identify safety, correctness or other mathematical features. Some theorem provers are fully automatic but in many cases interaction with proof assistants are required to complete the proof process.

The choice of suitable theorem provers is not an easy process. There are currently many theorem provers which can provide help in writing proofs. Some applications may require proofs to be written in detail while some requires complete automation. The work in this thesis attempts to compare three popular theorem provers which are Athena, Coq and Isabelle. We select representative examples from different domains and use each theorem prover to write formal proofs. During these steps, we compare these tools based on some criteria which include proof automation, readability, debugging, and documentation.

Keywords: Theorem prover, proof assistant, first-order logic, Athena, Coq, Isabelle.

TEOREM İSPATLAMA SİSTEMLERİNİN KARŞILAŞTIRILMASIYLA İLGİLİ BİR ÇALIŞMA

Adamu Sani YAHAYA

YüksekLisansTezi – Elektrik ve Bilgisayar Mühendisliği

Haziran 2014

Tez Danışmanı: Yrd. Doç. Dr. Aytekin VARGÜN

ÖZ

Teorem ispatlayıcıları ve ispat yardımcıları genellikle matematiksel problemleri çözmek için ya da yazılım ve donanımların formal olarak doğrulanmasında kullanılır. Yazılımların bakım işleminin yüksek maliyetli olmasını engellemek için müşterilere doğru ve güvenli kod teslim etmek kritiktir. Bu aynı zamanda vazgeçilmezdir çünkü sağlık, havacılık ve nükleer reaktörler ve benzeri alanlarla ilişkili uygulamalarda yaşamı tehdit eden problemlere neden olabilir. Teorem ispatlayıcıların kullanılmasıyla yapılabilen formal doğrulama, yüksek doğruluk ve güvenirliliğin garanti edilmesini sağlar. Teorem ispatlama süreci güvenilirlik, doğrululuk, ve diğer matematiksel özelliklerin ifade edilmesini (yazılmasını) ve bunlarla ilgili ispatların yazılması işlemini gerektirir. Bazı teorem ispatlayıcılar tamamıyla otomatiktir ancak birçok durumda ispatlama işleminin tamamlanabilmesi için kullanıcının sistemi yönlendirmesi gerekir.

Uygun teorem ispatlayıcının seçimi kolay bir işlem değildir. Halihazırda ispat yazmada yardım sağlayabilen pek çok teorem ispatlayıcı vardır. Bazı uygulamalar ispatların detaylı bir şekilde yazılmasına ihtiyaç duyarken diğer bir kısmı ise tam otomasyon gerektirir. Bu tezdeki çalışma, Athena, Coq, ve Isabelle isimli üç popüler teorem ispatlayıcıyı karşılaştırmayı amaçlamaktadır. Bunu yapabilmek için farklı alanları temsil eden örnek problemler seçtik ve bunlarla ilgili formal tanımlamalar ve ispatlar yazdık. Bu süreçte, ispatların otomatik olarak yazılabilmesi, okunabilirlik, hatalardan temizleme ve belgeleme gibi kriterleri kullanarak bu yazılımları karşılaştırdık.

Anahtar sözcukler: Teorem ispatlayıcılar,İspat yardımcısı,Birinci-derece mantık, Athena, Coq, İsabelle

DEDICATION

I dedicate this thesis to my parents for their tireless, indefatigable and earnest support, love, and encouragement they have been offering me since childhood to where and how I am today. My achievements in life wouldn’t have been materialized without their prayer, guidance and the philosophy of hardworking they imparted on me.

ACKNOWLEDGEMENT

Without the meaningful suggestions and advice from my supervisor, whose strenuous passion for software verification strengthens my enthusiasm for my research area, this thesis wouldn’t have gone beyond a mere dream. My sincere appreciation goes to my thesis supervisor and also course instructor Yrd. Doç. Dr. Aytekin VARGÜN

I remain grateful to His Excellency, the Executive Governor of Kano State-Nigeria, Engr. Dr. Rabi’u Musa. Apart from the knowledge acquired, this has given me an opportunity to explore other part of the world.

I also express my earnest indebtedness to my H.O.D and course instructor, Assoc. Prof. Ahmet Uyar together with the entourage of academic staffs of my department for all the support and the knowledge they imparted on me. This has contributed immensely while carrying out this research.

I similarly owe a great deal of thanks to my brothers, sisters, uncles, friends and most importantly 18 M.Sc students with which I benefited the 501 Kano State Scholarship Scheme.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

1.1 INTRODUCTION

In today’s world, there are many devices that are controlled by software in our daily life. Personal computers, airplanes, cells phones, washing machines and many similar devices have software on them. In many ways either directly or indirectly our lives, properties or our work depend on these devices. Their proper functionality is very important. There are several ways to check the accuracy of the software being used on them. One of the methodologies is to use formal verification which could be either automatic or semi-automatic. The role of automated reasoning or deduction is very important in formal methods and software verification. Automated reasoning and software verification reduce the amount of failure in software and improve the efficiency or performance. The theorem proving process requires one to write axioms, specifications, and any other related symbolic definitions first. In the next step, the user constructs proofs by using the language of the proof assistant. In some cases, the proofs can be obtained automatically. But in general, this process requires interaction from users who must have some expertise in proof engineering. In extreme cases, the whole proof might be written manually. There are many theorem provers which might have some degree of automation. In this thesis, we compare some of these theorem provers to help users to select a proper proof assistant for their needs.

1.2 Statement of the Problem

Coq, Athena and Isabelle are three popular theorem provers and proof assistants which have been used widely in the area in many projects. Although they have many

Similarities, they also have important differences. We observed that these theorem provers are not compared yet which is important since various problems require different approaches when proving theorems. The choice of right theorem prover that is related to the problem domain is extremely important since this choice can shorten the theorem proving process. As opposed to automated theorem proving, it might be necessary to present steps of proofs for mathematical problems.

Based on the literature review, this is the first work that compares Athena to Coq and Isabelle

In this research, we compare Athena, Coq and Isabelle by writing formal definitions and proofs which makes it possible to test correctness of some functions that we selected from different domains including natural numbers and lists. We selected five different representative examples and proved properties about them in all three theorem provers. One of the case studies we picked is to prove the equality of the definitions of factorial and tail-recursive factorial functions. These two functions are defined axiomatically in all three theorem provers and are proved that they are equivalent. We also defined a function called sum which returns the sum of natural numbers in a list. We used this definition as a specification and wrote definitions for a more efficient sum function. Our third example is selected as a mathematical problem which is based on sum of first n numbers. We also picked an example about intersection/parallelism of lines. Although this example was relatively simple, we collected valuable data from it where we needed to use proof by contradiction. Our final examples depict list operations. We defined and proved some properties of a function that reverses a list of natural numbers. All these examples required us to use existing libraries.

One of the goals of this research includes how easy it is to construct and maintain proofs and debug them by using a proof assistant. Are the proofs readable? Do the theorem provers have any documents and libraries to help users to write proofs. More specifically, we compared them based on the following criteria:

- Proof readability and understandability
- Bigger mathematical library

- Shorter proofs
- Flexibility in constructing proofs
- Better documentation
- Available on more platforms
- Better library lookup
- More feedback from the system
- Less meaningless typographic noise

These criteria are explained in Chapter-3 where we explain our methodology.

1.3 Organization of the Thesis

This thesis consists of 6 Chapters. In Chapter 2, we present previous work on automated theorem provers which are also compared by many researchers. The history of automated theorem provers and proof assistants is also discussed. Important terms in automated theorem proving are explained. The three theorem provers used in our research are introduced with a brief discussion to make it clearer for novice readers to understand our axioms and proofs. In Chapter 3, our approach was explained and some relatively simple proofs are illustrated in all three theorem provers. In Chapter 4, we present our case studies and explain proof steps while comparing the features of theorem provers. In Chapter 5, our results are illustrated and discussed in detail. We talked about the conclusions and possible future work in the last chapter.

CHAPTER TWO

LITERATURE REVIEW

2.1 RELATED WORK

The users of different mechanical proof-checking systems believed that there is little communication within them. The insufficiency of communication motivates them to give courage in preconception and myths about the ability to use variety of different systems. Tangible comparison shows clearly the advantages and limitations of various systems, in addition to their differences and commonalities. Furthermore, comparisons assist to find out what progress is being made and which way the field is heading.

Konstantine Arkoudas compared Athena and HOL theorem provers here [1]. In his research his main interest is to compare the simplicity and ease of expression: how easy or difficult to specify deductive processes that dynamically construct complicated proofs. He shows that in most situations Athena definitions and proofs are shorter. It is easier to express and understand Athena proofs. In his research he shows that Athena is efficient than HOL theorem prover.

Markus Wenzel and Freek Wiedjik in 2002 [2] compared Mizar and Isar theorem provers. The comparison determine a list of differences between Isar and Mizar, highlighted the strength of both systems from the views of end-users. In their Research they also show high-level views sketches in formal proofs. Wenzel and Wiedjik used just one example to compare the systems (the example is Euclid proof of the existence of infinitely many primes).

Freek Wiedjik, Dana Scott and Radboud in [3] also compared the style of different proof Assistants for mathematics which includes Mizar, HOL, Coq, Isabelle/Isar, PVS, Otter/Ivy, ACL2, Alfa/Agda, Theorema, Nuprl, IMPS, PhoX, Lego, Metamath, B method, -mega and Minlod. They used Pythagoras proofs of the irrationality of $\sqrt{2}$ in both formal and informal way in the mentioned systems to make their comparisons.

Vicent Zammit in [4] illustrated in his paper the differences between the style of theory mechanization of HOL and Coq. The comparison is based on the mechanization of fragment of the theory of computation in both HOL and Coq. Some examples are implemented to support the opinion debated in his paper. In addition, the mechanism for theorem proving and specifying definitions are discussed separately. They found that the strongest point given to the user is flexibility by means of metalanguage. Coq theorem prover relies on the power of calculus of Inductive construction. David Basin in [5] used examples to compare two systems which are Nuprl Proofs Development and Boyer-Moore Theorem prover. The example used is machine verification of a version of Ramsey theorem. Non-quantitative and quantitative measures were used to compare the proofs. Also in their Research they point-out that it is difficult to compare the naturalness and ease with which one finds proofs in different systems- especially when the theorem proving paradigms are different as NQTHM and Nuprl.

Our main focus is to compare Athena, Coq and Isabelle theorem provers. With our literature review we found out that Athena has not been compared with Coq and Isabelle theorem previously.

2.2 THEOREM PROVING AND THEOREM PROVERS

This section starts with a brief historical discussion of theorem provers and proof assistants. We then explain some common terms which include axioms, lemma, logic, proofs and theorems in general. We will show how to define or use them in all three theorem provers with a basic example in Chapter-3. We also provide a short introduction to theorem provers in general. We then specifically concentrate on the languages of Athena, Coq and Isabelle and explain them in more detail.

2.2.1 Brief History of Theorem Provers and Proof Assistants

In 1957 during class session Adridus wedberg called the Attention of his first year students where he made mention that proving Mathematical theorem in first-order logic on computer can be possible. This statement encourages his student Dag Prawitz to used the idea into practice [6]

The First general purpose machine or computer became obtainable after 2nd world war. In 1954, Martin Davis designed a program for advanced study about Presburger's algorithm using JOHNNIAC vacuum tube computer at Princeton Institute. Based on Martins, "Its great achievement was to prove that the sum of two even numbers is even" [7][8]. A .Simon, J.C. Shaw and Allen Newell developed a deduction system know as logic Theory machine (LTM) which "built proofs from sets of axioms and three rules of deduction: variable substitution, modus ponens, and the insertion of formulas by their definition". These systems make an effort to imitate human mathematician but do not guarantee to establish every valid theorem. They prove thirty eight out of fifty theorems in principia Mathematica [7].

Upon to proof assistants, in 1970's the idea of mathematical proofs with computer came up at different places at the same time [9]. The origin of proof assistant will be discussed here: Let us begin with Automath. The Automath project [10, 11] was started in the year 1967 by De-Bruijn whose goal was to developed a mathematical verification machine system. The related aims of the project was to design a language based on mathematics in which all the mathematical structures can be represented exactly, in the way where linguistic accuracy will give correct mathematical structures. This language has to be computer verifiable and supportive in enhancing their reliability and dependability in mathematical results. Various Automath systems have been used and applied in the mathematical formalization. Another essential initiative that first included in Automath is the Logical Frameworks. De-Bruijn stated that his system has only provided with the fundamental mathematical mechanisms in unfolding definitions, variable bindings, substitutions and creating etc. and an end-user can freely make any additions of logical rules as he/she wishes [9].

Martin-Löf extended the idea of curry-Howard formulas-as-types isomorphism which provides a connection of proofs between λ-calculus and constructive logics, designed constructive type theory as root for mathematical approaches, where functions and inductive types defined by understandable recursions are the basic principles [12, 13]. The first extended theory was used in Nuprl [14], Agda [15] and ALF [16] also in various proof assistants like Coq, LF and lego.

Logic for Computational Functions (LCF) was developed by Milner for formal theory which was defined by Scott in 1969. LCF born the systems HOL-light, Isabelle and HOL. They all used LCF approaches. The first system developed was a directed system that has tactics to disintegrate a larger goal into pieces of smaller goals. This system also included a simplifier that would simplify smaller goals and add a new proof command safely without storing proofs in memory but with the facts that are proved [8].

Trybulec built a system in 1973 at the University of Bialystok called Mizar System which was divided into two: Mizar Language and Mizar System. Its main goal is formal languages similar to mathematical languages where the system is a computer program which allows text file verification that is written in Mizar for mathematical correctness [9].

The first Nqthm was started in 1973 and arouse from the work of McCarthy. It was implemented in Lisp and is an automated reasoning system, it can be called “Boyer-Moore theorem prover”, and the interesting idea of Nqthm was concentrated toward automation and by combining interactiveness which allowed people to add lemmas. The logic behind is quantifier-free which is first-order logic with equality. The idea was used in Twelf Logical framework and evolved into ACL2 [9].

2.2.2 Theorem Provers

In computer science, theorem provers can be referred to as:

- Automated theorem provers which searches for a proof that a given statement is either true or false automatically or,
- Proof assistant which is an interactive theorem prover that requires a high degree of user interaction to find the proof

2.2.2.1 Automated Theorem Provers

Automated theorem proving (ATP), which is also called automated deduction, is a subdivision of mathematical logic. Automated reasoning deals with proving theorems of mathematics by a computer program. Proving mathematical expressions by using automated reasoning made a big impact on of computer science [17].

ATPs deal with the design of computer programs which show that a conjecture or in simplest word a statement is a logical outcome of a set of axioms or/and hypothesis. ATP plays vital roles in various places and areas such as mathematical centers, management consultants and hardware developers. For example, mathematicians can prove a statement that groups of order are commutative by applying the axioms of group theory; hardware developers can validate circuits by proving a statement that describes the equality of the circuit and its optimized version by using axioms and proof techniques; A management consultants can design a set of axioms which explain how community grows and interacts, and from those axioms it can be proved that community death rates increase or decrease with ages. All of these problems can be solved by the use of an ATP system with a proper designation of the problem which is defined with axioms, hypotheses, and conjectures.

2.2.2.2 Proof Assistants

In mathematics and computer science, interactive theorem prover or a proof assistant is software which provides help with the developments of formal proof by human-machine cooperation. This involves some sorts of interactive proof editor, with which an end-user can guide proof search, details of information are stored in, and a few steps are supplied by a machine (computer) [18].

A lot of automated proof systems began from complete automatic theorem prover to proof assistants. The initial proof assistants are based on first-order and propositional logic while the current ones are general-purpose systems which are identified as “proofs processor with spell-checkers.” which look similar to normal words-processors with a spelling checker that helps catching small errors while typing. Also proof assistants assist to developed proofs while removing errors and checking correctness [19].

The entire recent proof assistants are more or less influenced by Logic for Computable Functions (LCF)[20]. It introduced a functional meta-language that should be used to merge primitive reasoning or justification steps into extra sophisticated pattern of reasoning’s know as tactics. Its further fundamental contribution is to separate the center or core of the proof checker from the remaining part of the system. The latter guarantees that the correctness of the proof relied barely on the correctness of the proofs checker [19].

The most popular front-end for proofs assistant is Emacs-based editors, which was developed in the University of Edinburg. The proof assistants include coq, Isabelle and so on. The proof assistant for coq comprised of CoqIDE, which is based on OCaml/Gtk.

Isabelle comprised Isabelle/jEdit, which is based on jEdit and the Isabelle/Scala infrastructure for document-oriented proof processing [18].

Table 1: Comparison of proof assistants [3]

Proof Assistants	HOL	Mizar	PVS	Coq	Otter/Ivy	Isabelle	Alfa/Agda	ACL2	PhoX	IMPS
small proof kernel (‘proof objects’)	+	-	-	+	+	+	+	-	+	-
calculations can be proved automatically	+	-	+	+	+	+	-	+	+	+
extensible/programmable by the user	+	-	+	+	-	+	-	-	-	-
powerful automation	+	-	+	-	+	+	-	+	-	+
readable proof input files	-	+	-	-	-	+	-	+	-	-
based on higher order logic	+	-	+	+	-	+	+	-	+	+
large mathematical standard library	+	+	+	+	-	+	-	-	-	+

2.3 COMMON TERMS AND RELATED DEFINITIONS

2.3.1 Axioms

The word originated via Latin, from Greek ἀξίωμα (*āxīoma*) “that which is decision; thought fitting; self-evident principle”. Almost two and half decades of centuries ago, axiom was believed to be a statement that was absolutely true without a minimum amount of false suspicion, a self-evident truth. Nowadays non-mathematicians were convinced from mathematicians that an axiom is a rule in a game or only a premise and a point were ever rules begin [21].

In Mathematics, the term axiom is used in two associated but obvious senses "non-logical axioms" and "logical axioms" which will be described soon. In both senses, mathematical axioms are any statement in mathematics that serves as an initial point from which some others mathematical statements are derived using logical methods. Within the boundary of the system they define, axioms cannot be verifiable by mathematical proofs, nor derived by principles of deduction, basically for the reason that they are starting points. However, a Mathematical axiom in some systems may perhaps be a theorem in other systems, and vice versa.

Non-logical axioms (e.g. as in commutative over +, b + a = a + b) are truly defining properties belonging to the domain of mathematical theories specifically (such as arithmetic's) [22].

2.3.2 Theorem

The word theorem was originated from Greek called *theorema,* from the word *theorein meaning as* "to look at," of unknown origin. In mathematics and computer science, theorems are statements that have been proven on the origin of formerly established statement, like other theorems and normally accepted statements, for example like axioms. A mathematical theorem's proof is a logical argument for the statement given in the theorem which set in accord with the deductive system rules. The proof of a theorem is regularly explained or interpreted as the justification of the fact in the theorem's statement. In line with requirement that the Mathematical theorems can be proved, the idea of a theorem is primarily deductive, in distinction to the impression of a theory in science, which is called empirical [23].

Although theorems can be expressed in an absolutely symbolic forms, for e.g., in calculus proposition, theorems are frequently articulated in natural languages such as English language. The Same in proofs, which are frequently articulated as apparently worded and logically planned in informal arguments, with intention to satisfy readers about the truth in theorem statements away from any reasonable doubts, and through which symbolic proof might be constructed. These arguments are normally simpler and easier to verify than solely symbolic ones. Certainly, many mathematicians would explain in some way *why* proof is apparently true and also express a preference for a proof that not only demonstrates the validity of a theorem.

A Lot of mathematical theorems are based on conditional statements. In this sense, the proof extracts the conclusion from the hypotheses. In light of the interpretation of proof as justification of truth, the conclusion is frequently seen as a compulsory outcome of the hypotheses, that is, the validity of the hypotheses is true which will make the conclusion to be true, with no any additional assumptions. However, the condition could be explained and interpreted in a different way in some deductive systems, relying on the meanings given to the ruled of derivation and the conditional symbol [24].

2.3.3 Logic

The Name logic was originated from ancient of Greek word “logike” which has two meaning: the standard study of reasoning and secondly it described or explained validity of reasoning in various events [25]. Logic is rearrangement of facts to find the information that we want [26]. The first feature of logic was used significantly in the field of mathematics, philosophy and computer science; formal logic was developed date back to the period of ancient times in Greece, India and China. The study of logic is the development of valid inference in science, Greek logic; specifically the Aristotelian logic, its application was widely accepted in science and mathematics [24].

In this section we will talk about some types of logics and formal logical systems. The types of logic include propositional logic and predicate logic. In formal logical system will include first-order logic, other classical logic, non-classical logic and Algebraic logic.

2.3.3.1 Types of Logic

2.3.3.1.1 Propositional Logic

A propositional logic or propositional calculus is the lowest level or most restricted logical language where the basic unit of language is a proposition. The proposition is a statement which is either false or true. In simple statements, propositional logic is a formal system whereby some formulae representing a proposition can be created by a combination of atomic propositions with the help of some logical connectives. The established formal proof rules are known as “theorem”.

These logical connectives can be disjunctions (Logical OR) and conjunctions (logical AND) with the addition of negation of a proposition. There might be other appropriate symbols for some other statements such as implication which might be expressed in terms of conjunction, disjunction and negations. Logic does not bother with proposition by itself but their falsehood or truth hood. Proposition may perhaps be a statement like it may stand for a bit value in digital circuit or “All men are mortal” or and the logic will not be distorted [27].

A proposition is a well formed formula.

If A is a well formed formula then so is

¬A which is (NOT A).

If A and B are well formed formulae then so is the conjunction

A /\ B which is (A AND B).

The disjunction

A \/ B which is (A OR B);

And so on.

2.3.3.1.2 Predicate logic

Before going deeply into predicate calculus or logic, let us begin by defining predicate and subject. Subject is normally an individually entity, for example a boy or a school or a country. It may also be a class of entities, for example, all boys. A predicate is an attribute or property or mode of existence which a given subject may or may not have. For instance, an individual boy may or may not be skilful. Also a boy may or may not have brothers.

In predicate calculus or logic, the predicate/subject is different from the way it’s in Aristotelean logic. The main idea is, in predicate logic, a subject is an individual entity. For instance, an individual boy can be treated as a subject, but the class of all boys must be treated as a predicate [28].

Predicate logic provides an account of quantifiers general enough to express a wide set of arguments occurring in a natural language. Predicate calculus or logic is the generic term for symbolic formal systems such as first-order logic, second-order logic and infinitary logic [25]. More details will be discussed in the next part.

2.3.3.1.3 Formal Logical Systems

This section presents first-order, and other classical logic.

2.3.3.1.3.1 First-Order Logic

Propositional logic was extended to predicate logic in first-order logic. Whereby proposition is basically true or false, the truth value of functions of terms is the predicate that possibly be defined over domain or any non-empty sets. Variables, constant values or functions over the domain can be terms. The relationship between elements of the domain can be seen as predicates. A predicate of an arbitrary n defines the relations over product set of D^n. Moreover, a variable can be quantified by using universal quantifiers and existential quantifiers.

Conventional notation in first-order logics, predicates are represented as capital letters while variables, constants and functions are represented as lower case letters. These conventions are not fixed. They can be changed at anytime and in machine-based systems it is suitable to write all functions in prefix syntax for more readability. Some systems allow infix form to be used too [27]. Their syntax is very similar to propositional calculus but quantifiers are included.

2.3.3.2.2 Higher-order Logic

In higher-order logics quantification over predicates and function is allowed in addition to elements of the domain. A predicate is a function and a variable itself can also be a function which may be quantified over to express properties hold for the entire predicate and functions. In higher-order logic, set theory can be expressed in natural and direct way, in contrary to the convoluted method which needs to express every kind of set theory in first-order logics. First-order logic may perhaps express ZF which is known as zermelo-fraenkel set theory but it is very complex and difficult to show even easiest concepts like ordered pair. Therefore, in higher-order logic it ease the complexity in expressing the language by extending and improving the expressive power of the language. But it has the cost in terms of the degree and decidability to which the process of the proofs can be automated. A Complete and standard 2nd order logic is explained in details by Manzano [27].

2.3.4 Proofs

A proof is an argument for the truth of a proposition. In any mathematical field defined by its axioms or assumptions, a proof is arguments producing a theorem of that field through acknowledged inference rules which begin from either established theorem or those defined axioms. The topic of logic in specific proof theory studies and formalizes the idea of proof in formal approach [29][40-47].

2.3.4.1 Formal proofs

In a simple sentence a formal proof or a derivation is a sequence of finite sentences where each is an axiom which follows a sequence of rule of inferences. The final sentence in the sequence is a theorem in the formal system. Computers help in constructing formal proofs in an interactive theorem proving environment and these proofs can automatically be verified by computers.

2.4 ATHENA THEOREM PROVER

Athena language and/or proofs system is a programming language and interactive theorem prover environment which brought a way to show and works with mathematical proofs, so that they are both computer checkable and human readable. The language was developed by K. Arkoudas in MIT.

Athena language was designed to be a programming language which is a Higher-order functional language and also in many-sorted first-order logics based on the traditions of ML and scheme. It does encourage a programming techniques based on recursion and functions call with imperative characteristics [30].

Athena has primitive functions for mapping, unification, substitution and matching. ML-like pattern matching is provided for defining both functions and Method. Athena also offer support for both backward and forward inference steps, proofs by induction, proof by contradiction and equational reasoning [31].

With Athena it is doable to designed and write proof as function-like construction know as method whose executions can either results to Theorem or Lemma if inference rules is apply accurately, or an error conditions will be prompted which will makes it to stop if there are no rules of inferences is applied. Proofs can be represented at high-level abstraction, allowing the end-user to call the method and function defining it in various ways, for that reason the same functions or methods does not have to redeveloped for different parameters. With the same step generic library of proofs can be constructed [32].

2.4.1 Some Features in Athena

2.4.1.1 Terms in Athena

Terms can be either variables, functions application or constant, where functions can be written in the form (f $x_1, x_2 \ldots\ldots x_n$) and f is the function symbols while $x_1, x_2, \ldots.. x_n$ are the variables, constant or functions application. Each term must contain a type datatype or some domains.

Function symbols are declare to have precise range types and domain with declaration of the structure (declare f (→ (d_1, d_2, ….d_n) r)) which corresponded to normal notation of mathematics $f\colon d_1 \times d_2 \times \ldots\ldots\ldots \times d_n \rightarrow r$. Where constants are function without single argument, which declared as: (declare point (-> () Point)) or (declare point Point).

For example: if Nat is a datatype then (declare point (→ (Nat Nat) Point))

Where point is a function taking two natural numbers as arguments and it produce a result of another point, and (declare X_1 Nat) this show that X_1 is a constant of type Nat.

The type in function symbol definition must match with the related type argument in function application, and the range type of the function symbol is declared as the type application.

2.4.1.2 Proof Methods in Athena

In Athena we have various methods to prove a proofs such method includes proof by implication and equality chaining, proof by induction and proof by contradictions. Some proofs use two or more methods. The structure of each method will be shows and with some explanation about each proofs [32].

2.4.1.2.1 Equality and Implications chaining

Equality and Implications is one of the most useful approaches uses in each proofs methods. It proves equation by chaining together a sequence of terms connected by equalities. Athena expresses such methods as which can proves this proofs t_0 = tn as:

```
(! chain [t0 = t₁ [J₁] = t₂ [J₂] = · · · = tₙ [Jₙ]])
```

Where J_i is the justification of each step taken and it must be sentence inside the assumption base. The sentence can be axioms, or previous proved theorems.

Also in implications chaining, to prove the implication S_0 => S_n we can expressed it as:

(!chain [S_0 ⇒ S_1 [J_1] => S_2 [J_2] => · · · => S_n [J_n]]) where the Si are sentences and the justification J_i proves (S_{i-1} => S_i) .

2.4.1.2.2 Proof by Induction

Mathematical induction divides a proof of the form (∀n. (P n)) into two parts. The first part is the basis case while the other one is the inductive case:

(i) (P 0) and

(ii) (∀n. (P n) => (P n + 1)).

The property of natural numbers can be defined in Athena as a datatype as shown below:

```
datatype N := zero | ( S N)
```

We first defined natural numbers above. We could then prove properties of natural numbers by using some basic axioms and proof by induction rule whose structure is presented below:

```
by-induction ( forall ? n . P ?n) {
zero =>
   conclude (P zero )
        ...
| (S n) =>
     let { ind-hyp := (P n )}
        conclude (P (S n ))
               ... }
```

Where every induction principle begins with by-induction and the name “ind-hyp” means induction hypothesis (P n). The line that starts with zero is where we prove the basis case. In the definition of natural numbers we used the letter S which stands for the successor operator. The line that starts with (S n) is where we start proving the induction step.

2.4.1.2.3 Proof by Contradiction

Proof by contradiction can be expressed as:

```
(! by-contradiction P
             assume (~ P )     D)
```

```
(! by-contradiction P
             assume (~ P )
                (suppose-absurd p
                (!absurd R Q))
```

D is obtained as false with the call of the method called absurd “(!absurd R Q).” where R and Q have been deduced from (~ P), where Q is the negation of R. Some Athena proofs about the proof by induction and proof by contradiction methodologies will be illustrated in the next chapters.

2.5 COQ THEOREM PROVER

Coq is an interactive theorem prover and a formal proof management system. It provides formal languages to express mathematical definitions or assertions and executable algorithms. Mechanically checked proof of these assertions which help to find formal proofs and extract authorized programs from formal specifications which are developed from constructive proofs [33]. Coq was developed in France, by the following joint team called PI.R2. The teams included in the development are CNRS, Paris Diderot University, Ecole Polytechnique, INRIA, and Paris-Sud Univ. The Team was lead by Hugo Herbelinand implemented in Ocaml [34].

It is based on a very expressive logic, the Calculus of Inductive Constructions (CIC) which is originated within boundary of theory of calculus of inductive construction where it works. Coq is not an instrument that will automatically prove theorems but includes automatic theorem proving tactics and a variety of decision measures that greatly simplify the improvement of formal proofs [33].

2.5.1 Proof Methods in Coq

In order to develop propositions and to make sure their validity we must establish proofs. Subsequent to the method developed by R. Milner for LCF systems, backward reasoning with tactics [35] is used in this system. The typical method to developed proofs are called goal directed. Below we show how to prove theorems in Coq with a typical scenario [36]:

1. The user writes the statement to be proved by the commands either Lemma or Theorem, including the name of the conjecture for further reference
2. The system shows the formula to be proved, which also displays the goal under a horizontal line “============”
3. After that the command will be entered to disintegrate goal(s) into smaller goals.
4. List of formulas that need to be proved will be shown by the system
5. Step 3 to 5 will be repeated until no goal is displayed by the system.

The decomposition made in step 3 is known as tactics. A tactic transforms a goal into a set of subgoals such that solving these subgoals is sufficient to solve the original goal and these tactics reduce the numerical amount of goals. If there are no more goals to be decomposed then the proof is complete. The completed proofs need to be saved. They are stored when the user commands `Qed`. The effect of this command is to save a new theorem whose name was given at step-1. More details about tactics will be explained later when a proof example is implemented. The syntax of coq prover is shown below:

```
Theorem or Lemma name_of_theorem: proposition to be proved.
Proof. (Optional)
…….
Tactics
………
Qed.
```

2.6 ISABELLE THEOREM PROVER

Isabelle is a generic theorem prover or proof assistant which supports interactive reasoning in a variety of formal theories or logics. Rules are represented as propositions not as functions and proofs as in Coq. Isabelle provides an important proof methods for various first-order logics, constructive type theory, higher order logic and zermelo-fraenkel (ZF) set theory. Higher Order Logic (HOL) theorem prover is the predecessor of Isabelle theorem prover and it is based on small logical core which guarantees logical correctness. It was written in standard ML and has followed the style of LCF-style theorem prover.

Isabelle also has features such as automated reasoning tools that are efficient, like a tableaux prover and term rewriting engine, as well as different decision events. Isabelle has been used to formalize various theorems in computer science and mathematics [39].

2.6.1 Proof Methods in Isabelle

Isabelle is a directed goal system that has tactics to disintegrate down a larger goal into pieces of smaller goals and which also includes a simplifier that would simplify simple goals. There are many methods to disintegrate goals in Isabelle. We will discuss Structural Induction, Case Distinction, and Simplifications.
General skeleton of Isabelle proofs are shown below [38]:

```
theory T
imports B1…… Bn
begin
   declaration, definition, and proof
Theorem or lemma Name_theoremorLemma: "equations"
apply(Methods)
apply(Methods)
done
```

2.6.1.1 Structural Induction and Case Distinction

Induction is executed by "`induct_tac`" and induction works for any datatype. In some cases, induction is overkill and a case distinction over all constructors of the datatype suffices. This is performed by `“case_tac”[38]`. Examples will be shown in the next chapter.

2.6.1.2 Simplifications

In Isabelle and other systems, simplification is one of the central theorem proving tools and it simply means repeated substitutions of equations starting from left to right. The tool used is known as a simplifier which has term rewriting capability and the equations that are used mean rewriting rules. To bring about simplifications, the attributes [simp] declares lemma to simplification rules where the simplifier simplifies the lemma automatically. The simplification method has a general format as:

```
simp list_of_Modifiers
```

The attribute of the simplification in theorem can be turned off and on:

```
declare Lemma-name [simp]
```

and

```
declare Lemma-name [simp del],
add: list of theorem names
```

or

```
del: list of theorem names
```

or

```
only: list of theorem names
```
[38].

Example of simplification methods will be explained and shown in the next chapters.

CHAPTER THREE

METHODOLOGY

3.1 OUR APPROACH

In this research we compared Athena, Coq and Isabelle. We first selected proof problems from different domains. We defined function symbols axiomatically and write proofs about them in each theorem prover. For example, one of the examples that we selected is about the factorial of natural numbers. There are different ways of coding this to compute factorials. The regular recursive versions suffer from memory limits since factorials of big numbers require too many calls which will result in too many stack frames. If the same computation is done by using a tail-recursive version, the program will use only a single stack frame. This version would be more efficient and is therefore more preferable. We show that the two definitions are the same so one can use the tail-recursive version to compute factorials instead of the regular one. In this thesis we wrote formal proofs for doing this. We defined the following functions and proved the properties shown below in all three theorem provers:

- Equivalence of factorial and factorial efficient,
- Equivalence of a Sum-list and a tail recursive sum-list function which compute the sum of all natural numbers in a list in two different ways,
- Equivalence of Reverse and Reverse Efficient which reverse a list of natural numbers by using the regular and tail-recursive versions.
- Double reverse which applies the reverse function twice to the same list.
- A function that computes the sum of numbers from one to n and proved the related formula.
- Point and Line objects in geometry are defined. We then proved properties about the intersection and parallelism of lines.

While proving these theorems, we noted our experiences and compared these theorem provers based on the criteria we listed in Chapter-1. These will be explained in this chapter one by one now.

3.1.1 Proof readability and understandability

When we talk about readability, we mean something looking obviously clear to users. When the readability [37] is good the text can be read and understood easily. Readability can be determined if the definitions of symbols, axioms and theorems are understandable without spending too much time when reading them. The proofs should be understood without spending too much effort. This would make it possible for users to update the proofs and definitions when needed. Sometimes high-level definitions and functions do not make proofs look very clear.

3.1.2 Bigger mathematical library

A library is an organized collection of sources of information and similar resources, made accessible to a defined community for reference [51]. In this situation, bigger mathematical library simply means how big collection of organized theorems or lemmas are there inside the formal library for further reference. We have counted all the theorems in all three theorem provers to compare the size of the libraries roughly.

3.1.3 Shorter proofs

The term short simply means very few. In our perception when we said shorter proof we mean a proof that consists of a fewer line of code from the beginning of the proof to the end.

3.1.4 Flexibility in constructing proofs

Flexibility means the ability of changing, bending and modifying things without altering its meaning. Flexibility in constructing proofs is the ability of users to change or direct proofs to his desired direction without changing the meaning or breaking rules.

3.1.5 Documentation

Documentation is a set of documents provided on paper, or online, or on any other digital platforms. Examples are user guides, on-line help, and quick-reference guides and email lists. It is becoming less common to see paper (hard-copy) documentation. Documentation is distributed via websites, software products, and other on-line applications.

3.1.6 Platforms

In our point of view, Platform typically refers to a computer's operating system. For example: Windows, Unix, or Macintosh platforms and so on.

3.1.7 Library Lookup

A library lookup is a tool or command which used to search for a theorem, definition or lemmas inside a formal library especially larger library. It is very difficult to remember the names of all definitions, lemmas and theorems available in the current context, especially if the user has access to very large libraries. Also it is sometimes very difficult to search for needed lemma or theorem manually by using brute force techniques.

3.1.8 Feedback from the System

Feedback from the system means a process whereby the system provides vital information at every point during theorem construction. It will help to rectify errors or mistakes by the users.

3.1.9 Meaningless Typographic noise

Meaningless Typographic noise is a process whereby the theorem prover system supports less important characteristics that may lead to a lot of code lines and probably to slow down the loading time. This can also make it hard for user to understand the proofs.

3.2 TACTICS THAT ARE USED IN COQ AND ISABELLE THEOREM PROVERS

Coq Tactics

The coq tactics below are explained in [52] as:

- **contradiction** :solves the goal when False, or A and ¬A appear in the hypotheses
- **trivial**: tries very simple lemmas to solve the goal
- **auto:** searches in a database of lemmas to solve the goal
- **intuition**: removes the propositional structure of the goal then applies auto
- **simpl**: The tactic simpl simplifies all fixpoint definitions in the goal (which is sometimes too much, in which case it is recommended to prove the relevant equations as theorems and use them in a controlled way with the rewrite tactic).
- **rewrite:** The rewrite tactic by default replace all the occurrences of u in P(u).
- **assumption:** proves the goal if it is computationally equal to a hypothesis.
- **apply:** tell Coq which hypothesis to use. Eg: apply H1 [53].
- **intros**: This tactic transforms a proof state with a goal that involves logical implication, by moving the left-hand side of the implication into our set of hypotheses, giving it a name, and leaving the right-hand side of the goal[53].
- **ring:** is a generic tactic which deal with algebraic manipulations like associative and commutative rewriting in specific structures; the user can extend these tactics for new data-type and operations[52].
- **assert**: assert A ,Given a goal G, `assert A` creates a new goal A and adds A to the context in which we have to prove G. This is typically used in case we first want to prove an intermediate result A. NB: In case A is a compound proposition, like P x, it has to be put between brackets, assert (P x) [53].
- **Unfold:** unfold tactic is used to replace an identifier with the value to which it is bound. More precisely, `unfold t` unfolds the definition of t in the goal

Isabelle Tactics

The Isabelle tactics below are explained in [38] as:

Simp: Simp tactic is a rule which simplifies a sub-goal based on the definitions used in the theory. Datatype and primrec declarations and few others implicitly declare some simplification rule. Explicit definitions are not declared as simplification rules automatically. Almost all theorems can become a simplification rule. To use theorem or a lemma in simplification rule, we added the attribute [simp] to the definition of the theorem. Example: `lemma add_0_right [simp]: "m+0 = m".` The simplifier will transform theorem into equation.

- **Auto (**used as `apply(auto))` : This command tells Isabelle to apply a proof strategy called `auto` to all sub-goals. The `auto` command will try to apply a simplification rule to all sub-goals concurrently to solve our goals.
- **unfold:** As in coq, it is used to expand the definitional equation. The tactic `unfold` merely unfolds one or several definitions, as in `apply (unfold add_def).` This is can be useful in situations where `simp` does too much. Warning: unfold acts on all subgoals!

3.3 EXAMPLE PROOF IN ATHENA THEOREM PROVER

As it is mentioned in the previous chapter there are various methods to prove a theorem. One can use proof by induction, contradiction, equality or implication chaining or mixture of them. We will present an example in this section by using the proof by induction method to prove a simple property.

In Athena we need to define and load axioms and can use the theorems that are already proved. Here is how to define and assert axioms into the assumptions base:

```
assert right-zero    := (forall ?n . ?n + zero = ?n)
assert right-nonzero := (forall ?m ?n . ?n + (S ?m) = (S (?n + ?m)))
```

When the two axioms above are asserted we get the following messages from Athena saying that these are part of the assumption base which is a database of rules that are loaded or proved:

```
The sentence
(forall ?n:N
       (= (Plus ?n:N zero)
          ?n:N))
has been added to the assumption base.
The sentence
(forall ?m:N
       (forall ?n:N
         (= (Plus ?n:N
                    (S ?m:N))
            (S (Plus ?n:N ?m:N)))))
has been added to the assumption base.
```

We will now prove that the addition operation is commutative. In order to prove it we define the following property which assumes that addition is already defined. We must load these definitions beforehand. This property is an intermediate property which will be used in the proof of the commutativity property.

```
define left-nonzero    :=
  (forall ?n . (zero + ?n = ?n))
```

This property is proved by using proof-by induction as shown below:

```
define left-nonzero    := (forall ?n . (?n + zero = ?n))

by-induction left-nonzero {
       zero =>
       pick-any m
         (!chain [((S m) + zero)
                  --> (S m)           [right-zero]
                  <-- (S (m + zero))  [right-zero]])
     | (S n) =>
     let {induction-hypothesis :=
              (forall ?m . (S ?m) + n = (S (?m + n)))}
       pick-any m
         (!chain [((S m) + (S n))
              --> (S ((S m) + n)) [right-nonzero]
              --> (S (S (m + n))) [induction-hypothesis]
              <-- (S (m + (S n))) [right-nonzero]])}
```

And the result is:

```
Theorem: (forall ?n:N
                  (= (Plus zero ?n:N)
                     ?n:N))
```

Then we can use the above axioms and the left-nonzero property (which is a theorem now) to prove commutativity over addition. The proof is by induction again:

```
define commutative := (forall ?n ?m . ?m + ?n = ?n + ?m)
by-induction commutative {
       zero =>
         pick-any m
           (!chain [(m + zero)
                    --> m             [right-zero]
                    <-- (zero + m)    [left-zero]])
     | (S n) =>
         pick-any m
           let {induction-hypothesis := (forall ?m . ?m + n = n + ?m)}
           (!chain [(m + (S n))
               --> (S (m + n)) [right-nonzero]
               --> (S (n + m)) [induction-hypothesis]
               <-- ((S n) + m) [left-nonzero]])}
```

When these proofs are entered into Athena, the following theorem will be added into the assumption base:

```
Theorem: (forall ?n:N
                (forall ?m:N
                  (= (N.Plus ?m:N ?n:N)
                     (N.Plus ?n:N ?m:N))))
```

3.4 AN EXAMPLE PROOF IN COQ THEOREM PROVER

In Coq, there are various ways to prove theorems. We will discuss how to write proofs step by step by using an example. The way we prove depends on the method we use. Proofs in coq start with the keyword “Theorem” or “Lemma”. Here is an example:

```
Coq < Lemma pluso: forall x: nat, x = x + 0. or

Theorem pluso: forall x: nat, x = x + 0.
```

Each line in coq must end with a full stop or dot “.” Coq can execute the code in the corresponding line which ends with a dot which signifies the completeness of the code. In our example we will show the proof of x = x + 0 with structural induction.

`Coq < Lemma plusO: forall x: nat, x = x + 0`

This line of code will produce:

```
1 subgoal
============================
forall x : nat, x = x + 0
```

Then we can apply simple induction on n or `intros n; elim n.` to apply induction on the goals.

`Coq < intros x; elim x.`

This will yield the following result:

```
2 subgoals
x : nat
============================
0 = 0 + 0

subgoal 2 is:
forall x0 : nat, x0 = x0 + 0 -> S x0 = S x0 + 0
```

We need to simplify the basis step and inductive step, and then we apply "simpl" command to simplify the basis case as shown below:

```
Coq < simpl.
```

```
2 subgoals
x : nat
============================
0 = 0
subgoal 2 is:
forall x0 : nat, x0 = x0 + 0 -> S x0 = S x0 + 0
```

Then Coq tactic `auto` command can be applied when left-hand side and right-hand sides are equal. Or if you want to simplify the step that you are currently working on and remove all simplified statements in coq system then apply `reflexivity` tactic or auto command. They both serve for the same purpose.

```
Coq < auto.
```

After applying the command `auto`, it produced the result as follows:

```
1 subgoal
n : nat
============================
forall x0 : nat, x0 = x0 + 0 -> S x0 = S x0 + 0
```

Here, we have one remaining goal which is the one for the inductive step. We can apply the `simpl` command to simplify the statement and `auto` to remove the simplified statements.

```
Coq < simpl; auto.
```

The result obtained is: `No more subgoals`, meaning that there are no more goals to solve which mark the end of the proof. And we can conclude or save by entering the command "Qed."

```
intro x; elim x.
simpl.
auto.
simpl; auto.
plusO is defined
```

We will now present another proof which is the commutativity of addition. We can write it in mathematics like this: ∀ x y, x + y = y + x, which can be written in Coq as:

```
Coq < Lemma com_plus: forall x y: nat, x + y = y + x.
```

We will see the result as:

```
1 subgoal
============================
forall x y : nat, x + y = y + x
```

We have an option to choose which of the variable to apply the induction: either x or y. coq has the ability to combine many rules at once.

```
Coq < simple induction y; simpl; auto.
```

The following result will be produced:

```
1 subgoal
x : nat
y : nat
============================
forall y0 : nat, x + y0 = y0 + x -> x + S y0 = S (y0 + x)
```

The base case is succeed by the help of "`auto`" command but does not handle the inductive step. We need to apply `rewrite` to solve the inductive step.

We can apply "`intros m′ E` "which will replace each occurrence of th inductive variable with m′ in the expression E.

```
Coq < intros m′ E;
```

This yields the following result:

```
x : nat
y : nat
m′ : nat
E : m′ + x = x + m′
============================
 x + S m′ = S (m′ + x)
```

We apply `simpl`. command to simplify and then apply the `rewrite <- E`. where E is the hypothesis.

```
Coq < rewrite <- E.
```

We have now this result:

```
x : nat
y : nat
m′ : nat
E : x + m′ = m′ + x
============================
 x + S m′ = S (x + m′)
```

by applying `auto` it will yield "`no more subgoal`" showing that there is no more goal to solve and marked the end of the prove. Then we enter command `Qed` to save the proof.

```
Coq < Qed.
```

```
simple induction m; simpl; auto.
intros m’ E.
simpl.
auto.
 rewrite <- E.
 auto.
com_plus is defined
```

We showed the whole proof in the table above.

3.5 EXAMPLE PROOF IN ISABELLE THEOREM PROVER

There are various ways to prove theorems in Isabelle. We will discuss them step by step with the help of an example. How to prove theorems depend on what process you want to use to prove. To start any proofs in Isabelle we must begin with the keyword “Theorem” or “Lemma” here is an example:

To prove the theorem “lemma "m + n = n + m"” which is called the commutativity over addition; we need some lemmas to be proved before proving the main theorem. The command “lemma” starts the proofs. While the apply (rule induct) instruct Isabelle to start proofs by induction.

```
lemma "m + n = n + m"
apply (rule induct)
```

This command yields the result:

```
proof (prove): step 1
goal (2 subgoals):
 1. m + n = 0
 2. ⋀x. m + n = x ⟹ m + n = Suc(x)
```

The numbered lines are called subgoals where the first one is the base case and the last one is the inductive step case. We used `back` command to calls back one step of the subgoal.

```
goal (2 subgoals):
 1. 0 + n = n + 0
 2. ⋀x. x + n = n + x ⟹ Suc(x) + n = n + Suc(x)
```

Applying `oops` will stop the proof of commutativity because we need another intermediate proof to continue with the actual proof. And that intermediate property to be proved is named as add_0 which is `0 + n = n`. The proof for this is shown in the table below:

```
lemma add_0 [simp]: "0+n = n"
apply (unfold add_def)
apply (rule rec_0)
done
```

We prove another property which is add_Suc. In the proof steps we apply `apply(unfold add_def)` which expands addition property from the definition stated and rule rec_0 concluded the proofs. In the add_suc proof `"Suc(m)+n = Suc(m+n)"` we also apply unfold and rec_suc to solve our proof.

```
lemma add_Suc [simp]: "Suc(m)+n = Suc(m+n)"
apply (unfold add_def)
apply (rule rec_Suc)
done
```

```
lemma add_0_right [simp]: "m+0 = m"
apply (rule_tac n = m in induct)
apply simp
apply simp
done

lemma add_Suc_right[simp]: "m+Suc(n) = Suc(m+n)"
apply (rule_tac n = m in induct)
apply simp_all
done
```

The lemma add_0_right proves "`m + 0 = m`". in this case we apply proof by inductions m. where we have two sub-goals, the zero case and inductive steps. We also apply "apply unfold" which expand addition definition and solve the zero case and then we solve the second sub-goal which inductive steps by apply rec_suc. In all our lemma we insert `[simp]` attribute which add the lemma into the simplifier. Whenever simp is apply then the lemma cam be apply if the sub-goal required it.

After proving all these lemmas we can continue to prove our main goal. Remember that our proof has stopped when we entered oops. Note also that we named the theorems after proving them for further references.

```
lemma add_com :"(m + n) = (n + m)"
```

and it yields this result:

```
goal (1 subgoal):

   1. m + n = n + m
```

```
apply(rule_tac n = m in induct)
```

Also yields the following result:

```
goal (2 subgoals):
 1. 0 + n = n + 0
 2. ⋀x. x + n = n + x ⟹ Suc(x) + n = n + Suc(x)
```

We apply: simp command with the help of the above lemma that had been proved

“apply simp”

Now the result becomes:

```
goal (1 subgoal):

   1. ⋀x. x + n = n + x ⟹ Suc(x) + n = n + Suc(x)
```

We also need to apply simp again to simplify the inductive base case and it yields “`no subgoals`” meaning that the proof is done or completed.

Here is the complete proof:

```
lemma add_com :"(m + n) = (n + m)"
apply(rule_tac n = m in induct)
apply simp
apply simp
done
```

CHAPTER FOUR

CASE STUDIES

4.1 CASE STUDY: EQUIVALENCE OF FACTORIAL AND TAIL-RECURSIVE FACTORIAL

In this section we first define factorial and tail-recursive factorial functions axiomatically in all three theorem provers. In the next step we show that these definitions are equivalent. Since the tail-recursive functions are more efficient, they are preferable to regular functions. Therefore one can use the tail recursive version if the equivalence relation is proved. Table below shows code for intuitive implementation and efficient implementation for factorial in java.

Definition of Factorial	
$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0. \end{cases}$	
Intuitive implementation	**Efficient implementation**

Intuitive implementation

```
public class Factorial {

    // Version-1: This version is
not efficient
    public static int
Factorial(int n){
        if (n==0)
            return 1;
        else
            return n*Factorial(n-
1);
    }
```

Efficient implementation

```
public static int
FactorialEfficient(int n){
        return
FactorialEfficient_Helper(n,1);
    }
    public static int
FactorialEfficient_Helper(int n,
int res){
        if (n==0)
            return res;
        else
            return
FactorialEfficient_Helper(n-1,
n*res);
    }
```

4.1.1 Equivalence of the Factorial and Tail-Recursive Factorial in Athena

In this example, two different factorial functions will be defined axiomatically. The factorial function takes a natural number as input and returns its factorial. We can define a regular factorial function as follows in Athena:

```
(declare Factorial (-> (Nat) Nat))


(define Factorial-zero-axiom
            (= (Factorial zero)
               (succ zero)))


(define Factorial-succ-axiom
   (forall ?x
      (= (Factorial (succ ?x))
         (Times (succ ?x) (Factorial ?x)))))


(assert Factorial-zero-axiom)
(assert Factorial-succ-axiom)
```

We first defined it symbolically. Then we stated two axioms that define the factorial. The first axiom states that the factorial of zero is one. The second one states that the factorial of (x+1) is equal to (x+1)*x!.

A function that is generated from these axioms is a recursive one which may have a stack-overflow problem when its input is too large. An efficient factorial function can be implemented to overcome the high usage of many stack frames which leads to overflow. The solution is to define a tail-recursive factorial function which uses only one stack frame during the execution.

The efficient version uses a helper function. Therefore we need to define two functions in Athena. Here is how we define the helper function:

```
declare FactorialEfficientHelper (-> (Nat Nat) Nat))


(define FactorialEfficientHelper-zero-axiom
   (forall ?res
      (= (FactorialEfficientHelper zero ?res)   ?res)))
```

```
(define FactorialEfficientHelper-succ-axiom
    (forall ?n ?res
       (= (FactorialEfficientHelper (succ ?n) ?res)
          (FactorialEfficientHelper ?n (Times (succ ?n) ?res)))))

(assert FactorialEfficientHelper-zero-axiom )
(assert FactorialEfficientHelper-succ-axiom )
```

We now can define the FactorialEfficient which uses the definition of the helper function that is defined above.

```
(declare FactorialEfficient (-> (Nat) Nat))

(define FactorialEfficient-axiom
      (forall ?n
            (= (FactorialEfficient ?n)
               (FactorialEfficientHelper ?n (succ zero)))))

 (assert FactorialEfficient-axiom)
```

After defining and asserting the definitions of factorial and it counterpart i.e. the factorial efficient as our axioms, we define a correctness statement as a conjecture as it is done below:

```
define Equality :=
      (forall ?n .(Factorial ?n) = (FactorialEfficient ?n))
```

This statement cannot be asserted as we did with axioms. We have to provide a proof of equivalence. We will now show the proof.

In some cases, the proofs are too large and may require an intermediate conjecture to be proved before we prove the main statement. These smaller goals are called lemma when we prove them. Here is the proof of the correctness statement that states the equivalence of the regular factorial and its more efficient version.

```
by-induction Equality{
  zero =>
      (!chain [(Factorial zero)
            -->(S zero)
               [Factorial-zero-axiom]
            <--(FactorialEfficientHelper zero (S zero))
               [FactorialEfficientHelper-zero-axiom]
            <--(FactorialEfficient zero)
               [FactorialEfficient-axiom]])
| (S n) =>
    let {induction-hypothesis :=
           (Factorial n) = (FactorialEfficient n)}
      (!chain
      [(Factorial (S n))
        -->((S n)*(Factorial n))
            [Factorial-succ-axiom]
        -->((S n)*(FactorialEfficient n))
           [induction-hypothesis]
        -->((S n)*(FactorialEfficientHelper n (S zero)))
           [FactorialEfficient-axiom]
       <--(FactorialEfficientHelper n ((S n)*(S zero)))
           [....]
      <--(FactorialEfficientHelper (S n) (S zero))
         [FactorialEfficientHelper-succ-axiom]
      <--(FactorialEfficient (S n))
         [FactorialEfficient-axiom]])
}
```

The proof given in the previous page uses a lemma which is stated and proved below. Any lemmas need to be proved before they are used in the proofs.

```
define P1 :=
      (forall ?n ?a .
            (FactorialEfficientHelper ?n ( ?a * (S zero)))
             = (?a * (FactorialEfficientHelper ?n (S zero))))
```

The proof of lemma P1 is shown below:

```
by-induction P1{
  zero =>
    pick-any a
      (!chain [(FactorialEfficientHelper zero (a*(S zero)))
            --> (a*(S zero))
            [FactorialEfficientHelper-zero-axiom]
            <-- (a*(FactorialEfficientHelper zero (S zero)))
            [FactorialEfficientHelper-zero-axiom]])
| (S zero) =>
    let {induction-hypothesis :=
         (forall ?a .
              (FactorialEfficientHelper n (?a*(S zero)))=
              (?a*(FactorialEfficientHelper n (S zero))}
      pick-any a
        (!chain
      [(FactorialEfficientHelper (S n) (a*(S zero)))
      -->(FactorialEfficientHelper n ((S n)*(a*(S zero))))
       [FactorialEfficientHelper-succ-axiom]
      <--(FactorialEfficientHelper n (((S n)*a)*(S zero)))
       [Times-Associativity]
      -->(((S n)*a)*(FactorialEfficientHelper n (S zero)))
       [induction-hypothesis]
      -->((a*(S n))*(FactorialEfficientHelper n (S zero)))
       [Times-Commutativity]
      <--(a*((S n)*(FactorialEfficientHelper n (S zero))))
       [Times-Associativity]
<--(a*(FactorialEfficientHelper n ((S n)*(S zero))))
   [induction-hypothesis]
      <--(a*(FactorialEfficientHelper (S n) (S zero))))])
       [FactorialEfficientHelper-succ-axiom]])
}
```

4.1.2 Equivalence of the Factorial and Tail-Recursive Factorial in Isabelle

As stated in Chapter-2, every theory in Isabelle must begin with this statement "theory name_of_the_theory" which is the name of the theory. After writing the name of the theory we proceed with a `imports` statement which helps us to use any existing theorems and definitions that are written previously. In the example below:

"import Nat",

The definitions and the theorems in the library "Nat" will be available. We can use them to write new proofs. As a beginner it is advisable to always use "main" as your imported theory unless you know the proof steps exactly. The declarations ends with a begin statement. After that point we write the actual proof. Here is an example:

```
"theory Equalityfact
imports Main Nat Rings
begin"
```

In Isabelle theorem prover the keyword "primrec" indicates a primitive recursive definition which is a definition that consists of equations. Each equation corresponds to one of the constructors for the datatype. In the examples below the "Fact" in "primrec" definition is the name of the function. "nat -> nat" means that definition "Fact" have one input as a natural number and returns another natural number.

```
theory Equalityfact
imports Main Nat Rings
begin
primrec Fact::"nat => nat"
where
  "Fact         0  = 1"
| "Fact (Suc n) = Suc n * Fact n"
```

In factorial_helper, the “nat -> nat ->nat” means that function factorial_helper takes two natural numbers as input and returns another one as an output. Below is definition of factorial_helper in Isabelle:

```
primrec Factorial_helper::"nat => nat => nat"
where
  "Factorial_helper        0  a = a"
| "Factorial_helper (Suc n) a = Factorial_helper n (Suc n * a)"
```

We will now write proofs for several conjectures/properties. In Isabelle, we don’t need to define the property separately. Rather we define it together with the proof which is not the same as in Athena where the definition of any property must be done before any proof begins. In this proof we apply proof by induction on n to prove our property. Here is the intermediate property to be proved:

```
theorem Equality:"Fact n = Factorial_helper n 1"
```

Here is the proof for this property:

```
theorem Equality:"Fact n = Factorial_helper n 1"
apply(induct_tac n)
apply simp
apply simp
done
```

The first “apply simp” tries to simplify our zero case sub-goals based on our “primrec” definition equation and was solved successfully. Because we have all we need in the primrec definition. Our second “apply simp” failed because the required definition or lemma is not in primrec definitions and sub-goal fails. We need to feed the simplifier the required lemmas to solve our sub-goal. That means we need an intermediate lemma.

```
theorem Equality:"Fact n = Factorial_helper n 1"
apply(induct_tac n)
apply simp
apply simp

done
```

During our proof steps we also prove the intermediate lemma which connects our proof steps. We proved the intermediate lemma with proof by induction technique. During the proof steps we found that we need other lemmas such as commutativity and distributivity over addition to complete our proof. We imported them from “theory ring” and add them into the simplifier to solve our sub-goal. Whenever such lemmas are needed they are used by the simplifier to complete the goals. The complete proof is shown below:

```
lemmas [simp] =
  ring_distribs
  diff_mult_distrib diff_mult_distrib2 --{*for type nat*}

theorem Equality:"Fact n = Factorial_helper n 1"
apply(induct_tac n)
apply simp
apply simp
done

lemma strong_fact[simp]:"ALL a. Factorial_helper n a = a * Fact n"
apply(induct_tac n)
apply simp
apply simp
done
```

4.1.3 Equivalence of the Factorial and Tail-Recursive Factorial in Coq

In Coq, the “Require Import” stated in the example below is an optional statement which is different from Isabelle where a proof cannot begin without an import statement. In Coq it can only be used when it is necessary. The function of the “Require

Import" is to extend the library you are concurrently using during the proof process. The term "Fixpoint" is used to define functions over recursive datatype. We use Fixpoint when trying to define a recursive function. In the definition of factorial which is represented by the symbol fact we state that it takes a natural number as an input and returns another natural number. Here is the Coq definition:

```
Require Import Arith ArithRing.

Require Ring.


Fixpoint fact(n:nat): nat :=

match n with

|O => S O

|S n => S n * fact n

end.
```

We also define helper_fact is tail-recursive factorial which accepts two parameters and returns a natural number. Below is definition in Coq:

```
Fixpoint helper_fact(n a:nat): nat :=

match n with

|O => a

|S n => helper_fact n (S n * a)

end.


Definition fn_fact (n:nat) := helper_fact n 1.
```

In Coq we define the property and prove it together. In the proof below, we use proof by induction on n to prove the Equality_fact property. For more details about the functions of rewrite, unfold, simp, auto please refer to the previous chapter. The proof steps are very similar to the ones in Athena and Isabelle where we need to prove an intermediate lemma (in this case we named it fact_eq_strong) to connect our proofs.

```
Lemma Equality_fact(n:nat):fact n = fn_fact n.

induction n.

rewrite fact_zero.
```

```
unfold fn_fact.

simpl.

auto.

unfold fn_fact.

rewrite fact_eq_strong.

simpl.

rewrite plus_0_r.

auto.

Qed.
```

We proved the intermediate lemma using proof by induction on ‘n’. We then used this lemma to complete the proof of the main theorem. The proof is shown below:

```
Lemma fact_eq_strong:forall n a,  helper_fact n a = a * fact n.

induction n.

simpl.

intros.

ring.

auto.

intros.

simpl.

rewrite IHn.

rewrite mult_plus_distr_l.

ring.

Qed.
```

4.2 CASE STUDY: SUM OF N NUMBERS

In this case study, we show a proof of sum of n numbers with all the three theorem provers. From these proofs, it is clearly shown that Athena definitions and proofs look much more readable. Proof steps are close to a hand-written proof in Athena compared to Isabelle and coq proofs and definitions. In Isabelle and Coq some steps are skipped due to its semi-automatic proof evaluation procedures. The function of sum of natural numbers can be defined as the sum of numbers from 1 to n (inclusive). In this section we proved the sum of n numbers in Athena and Isabelle and we used an existing solution of coq which is taken from [50]. The sum function takes one input and returns the sum of the numbers in the sequence from one to the input given. We show the proofs below:

4.2.1 Sum of n Numbers in Athena

Axiomatic definition of sum function is illustrated below. This function takes a natural number as input and returns the summation of the first n numbers. Below is definition of sum in Athena:

```
load "nat-times.ath"

datatype N := zero | (S N)

declare  Sum: [N] -> N
```

Here are the axioms that define this function:

```
declare one, two: N

assert one-definition := (one = (S zero))

assert two-definition := (two = (S one))

assert one-one-definition := (two = one + one)

assert n-one-definition := ( forall ?n . ?n + one = (S ?n))

assert Sum-zero-axiom := ((Sum zero) = zero)

assert Sum-succ-axiom :=

       (forall ?n .  (Sum (S ?n)) = (S ?n) + (Sum ?n))
```

Here is an intermediate lemma which we define and proved as the connector to complete the proof of sum of n numbers.

```
define fact-distr :=

        (forall ?z ?x ?y .  ?z * ?x + ?z * ?y = ?z * (?x + ?y))

by-induction fact-distr {

   zero =>

    pick-any x y
```

```
     (!combine-equations

     (!chain [(zero * x + zero * y)

     --> (zero + zero * y)              [N.Times.left-zero]

     --> (zero + zero)                  [N.Times.left-zero]

     --> zero                           [N.Plus.left-zero]])

   (!chain [(zero * (x + y))

   --> zero                             [N.Times.left-zero]]))

   | (S z)  =>

      let {induction-hypothesis :=

             (forall ?x ?y .  z * ?x + z * ?y = z * (?x + ?y))}

 pick-any x y

 (!combine-equations

 (!chain [((S z) * x + (S z) * y)

      --> ((x + z * x) + (S z) * y)  [N.Times.left-nonzero]
--> ((x + z * x) + (y + z * y))      [N.Times.left-nonzero]

  --> (x + (z * x + (y + z * y)))    [N.Plus.associative]

 --> (x + ((y + z * y) + z * x ))    [N.Plus.commutative]

--> (x + (y + (z * y + z * x)))      [N.Plus.associative]

--> (x + (y + (z * x + z * y)))      [N.Plus.commutative]

])

(!chain [((S z) * (x + y))
--> ((x + y) + z * (x + y) )             [N.Times.left-nonzero]
--> ((x + y) + (z * x + z * y))          [N.Times.left-distributive]

--> (x + (y + (z * x + z * y)))          [N.Plus.associative]

]))

}
```

After defining and asserting the definition of sum of n numbers, we define the sum of n number conjecture which is shown below:

```
define N-Sum := (forall ?n . two * (Sum ?n) = ?n * (?n + one))
```

This statement cannot be asserted as we did with axioms. We have to provide a proof for this formula. We will now show the proof.

```
by-induction N-Sum {
  zero =>
   (!combine-equations
   (!chain [(two * (Sum zero))
        --> ( two * zero)           [Sum-zero-axiom]
        -->  zero                   [N.Times.right-zero]])
   (!chain [(zero * ( zero + one))
       --> zero                     [N.Times.left-zero]]))
| (S n) =>
  let {induction-hypothesis := (two * (Sum n) = n * (n + one))}
  (!chain [(two * (Sum (S n)))
       --> (two * ((S n) + (Sum n)))           [Sum-succ-axiom]
       --> (two * (S n) + two * (Sum n))       [N.Times.left-
distributive]
       --> (two * (S n) + n * (n + one))       [induction-hypothesis]
       --> (two * (S n) + n * (S n))           [n-one-definition]
       --> ((S n) * two + n * (S n))           [N.Times.commutative]
       --> ((S n) * two + (S n) * n)           [N.Times.commutative]
       --> ((S n) * (two + n))                 [fact-distr]
       --> ((S n) * ((one + one) + n))         [one-one-definition]
       --> ((S n) * (one + (one + n)))         [N.Plus.associative]
       --> ((S n) * (one + (n + one)))         [N.Plus.commutative]
       --> ((S n) * (one + (S n)))             [n-one-definition]
       --> ((S n) * ((S n) + one))
[N.Plus.commutative]])}
```

4.2.2 Sum of n Numbers in Coq

An axiomatic definition of sum function in Coq is illustrated below.

```
Require Import Arith ArithRing.

Require Ring.

Ltac defn x := unfold x; fold x.
```

```
Fixpoint sum (n : nat) : nat :=
 match n with
  | O => O
  | S n => S n + sum n
 end.
```

Now we are going to prove the mathematical formula that is presented in the previous section. The proof is by induction over ‘n’. See the previous chapter for the functions of trivial, ring, etc.

```
Theorem sum_equals: forall n, 2 * sum n = n * (n + 1).


induction n.
trivial.
defn sum.
rewrite mult_plus_distr_l.
rewrite IHn.
rewrite mult_comm.
ring.
Qed.
```

4.2.3 Sum of n Numbers in Isabelle

An axiomatic definition of sum function in Isabelle is presented below.

```
theory sum
imports Main Rings
begin
primrec sum :: "nat => nat" where
"sum 0 = 0" |
"sum (Suc n) = (n + 1) + sum n"
```

We will now show the proof of the same conjecture that is proved in Athena and Coq. Again the proof is by induction over ‘n’ to solve our sub-goals. See the previous chapter for the definition of simp tactic. Here are the definition and proofs:

```
lemma " 2 * sum n  = n*(n + 1)"

apply(induct_tac n)

apply(simp)

apply(simp)

done
```

4.3 CASE STUDY: SUM-LIST

An axiomatic definition of a function called sum-list will be presented in this section. The function sum list takes a list of natural numbers and returns the sum of the numbers in the list. The Athena Sum-list proof was defined in the prefix notation in [49]. In this study, we only proved it in Coq and Isabelle. This example is also used for showing the proof process for a correctness property. The sum-list function can be defined in two different ways with different symbols. In the first version we define a regular recursive function which may consume the stack frames with repeated calls. The second version is more efficient since it is the tail-recursive version of the same function.

4.3.1 Sum-List in Athena

In order to prove the correctness of sum-list in Athena, we defined the sum-list function in two different ways. The first version represents a regular recursive function which may consume the stack frames with repeated calls. The second version is tail-recursive and thus more efficient. One can use the first definition as the specification of how to compute the sum of numbers in a list. In the next step we will just need to prove the equivalence of this specification and the definition of the tail-recursive version. The definitions are shown below:

Definition of sum-list@ (This is not efficient and can serve as the specification):

```
define sum-list@-empty :=  ((sum-list@ nil) = zero)

define sum-list@-nonempty :=

      (forall ?x:N ?L:(List N) .

       (sum-list@ (?x :: ?L)) = (?x + (sum-list@ ?L)))
```

Definition of sum-list (This is the more efficient version):

```
define sum-list-empty :=

  ((sum-list nil) = zero)

define sum-list-nonempty :=

       (forall ?L:(List N) ?x:N .

            (sum-list (?x :: ?L))=

          =(sum-list-compute ?L ?x))
```

Definition of sum-list-compute which is the extension of sum-list:

```
define sum-list-compute-empty :=

                  (forall ?x .

                   (sum-list-compute nil ?x)

                  = ?x);;

define sum-list-compute-nonempty :=

       (forall ?L ?x ?y .

             (sum-list-compute (?y :: ?L) ?x)

          = (sum-list-compute  ?L (?x + ?y)))
```

Below is the definition of sum-list-compute-relation. This intermediate property needs to be proved first. The proof is presented below.

```
define sum-list-compute-relation :=

      (forall ?L ?x .

             (sum-list@ (?x:N :: ?L:(List N)))

             = (sum-list-compute ?L ?x))
```

The definition of sum-list-correctness proof shown below is the actual correctness condition. If the proof is obtained, one can use the more efficient version instead of the non-tail-recursive function.

```
define sum-list-correctness :=

  (forall ?L:(List N) .

        (sum-list ?L) =

        (sum-list@ ?L))
```

Below is the proof of sum-list-compute-relation and sum-list-correctness:

```
define sum-list-compute-relation :=
      (forall ?L ?x .
             (sum-list@ (?x:N :: ?L:(List N)))
             = (sum-list-compute ?L ?x));;
by-induction
 sum-list-compute-relation {
 nil =>
  conclude (forall ?x:N .
              (sum-list@ (?x:N :: nil:(List N)))
             = (sum-list-compute nil ?x))
    pick-any x:N
      (!chain
       [(sum-list@ (x :: nil:(List N)))
       --> (x + (sum-list@ nil))     [sum-list@-nonempty]
       --> (x + zero)                [sum-list@-empty]
       --> x                         [N.right-zero]
       <-- (sum-list-compute nil x)        [sum-list-compute-empty]
       ])
| (y :: L) =>
  conclude (forall ?x:N .
               (sum-list@ (?x :: (y :: L)))
             = (sum-list-compute (y :: L) ?x))
    let {induction-hypothesis := (forall ?x:N .
                            (sum-list@ (?x :: L))
                             = (sum-list-compute L ?x))}
      pick-any x:N
      (!chain
       [(sum-list@ (x :: (y :: L)))
       --> (x + (sum-list@ (y :: L)))     [sum-list@-nonempty]
       --> (x + (y + (sum-list@ L)))      [sum-list@-nonempty]
       <-- ((x + y) + (sum-list@ L))      [N.associative]
      --> (sum-list@ ((x + y) :: L))      [sum-list@-nonempty]
      <-- (sum-list-compute L (x + y))    [induction-hypothesis]
      <-- (sum-list-compute (y :: L) x)   [sum-list-compute-nonempty]
      ])
};;

define sum-list-correctness :=
  (forall ?L:(List N) .
        (sum-list ?L) =
              (sum-list@ ?L));;


by-induction sum-list-correctness{
 nil =>
  (!chain
   [(sum-list nil:(List N))
   --> zero             [sum-list-empty]
   <--(sum-list@ nil) [sum-list@-empty]])
 | (x:N :: L:(List N)) =>
  (!chain
   [(sum-list (x :: L))
   --> (sum-list-compute L x) [sum-list-nonempty]
   <-- (sum-list@ (x :: L))   [sum-list-compute-relation]])};;
```

4.3.1 Sum-List in Isabelle

In order to prove the correctness of sum-list in Isabelle, we defined the sum-list function in two different ways as stated in Athena. The imported theories are "Datatype" and "Main" which are shown below. The proof is based on the imported theory and the datatype of the list shown below:

```
theory Sumlist

imports Datatype  Main

begin

datatype 'a list = Nil                     ("[]")

               | cons 'a "'a list"        (infixr "#" 65)
```

Definition of sum-list' (This is not efficient and can serve as the specification):

```
primrec sum_list' :: "nat list  => nat" where

"sum_list' []  = 0"

| "sum_list' (x#xs)  = x + sum_list' xs "
```

Definition of sum-list (This is the more efficient version):

```
primrec sum_list :: "nat list  => nat" where

"sum_list []  = 0"

| "sum_list (x#xs)  = sum_list_compute xs x "
```

Definition of sum-list-compute which is the extension of sum-list:

```
primrec sum_list_compute :: "nat list => nat => nat" where

"sum_list_compute [] y = y"

| "sum_list_compute (x#xs) y = sum_list_compute xs (y + x)"
```

The definition of sum-list-correctness and its proof are shown below. This is the actual correctness condition. If the proof is obtained, one can use the more efficient version instead of the non-tail-recursive function. Also the proof of the intermediate lemma "sum_list_compute_relation" is shown together:

```
theorem   sum_list_compute_relation[simp]:"   sum_list_compute   Ls   x   =
sum_list'(x # Ls)"

apply (induct Ls arbitrary: x)

apply (simp_all)

done

theorem sum_list_correctness: " sum_list L = sum_list' L"

apply (induct_tac L)

apply (auto)

done
```

4.3.1 Sum-List in Coq

In order to prove the correctness property for sum-list in Isabelle, we did the same thing that are done for Athena and Isabelle

We should first import "Arith" and "ArithRing" theories as shown below.

```
Require Import Arith ArithRing.

Require Ring.

Inductive list : Set :=

 | nil : list

  | cons: nat -> list -> list.
```

Definition of sum-list' (This is not efficient and can serve as the specification) :

```
Fixpoint sum_list' (ls1 : list) {struct ls1} : nat :=

  match ls1 with

    | nil => 0

    | cons h ls1 => h + sum_list' ls1

End
```

Definition of sum-list (This is the more efficient version):

```
Fixpoint sum_list_compute (ls1:list ) (n:nat){struct ls1}  : nat :=
  match ls1 with
    | nil  => n
    | (cons h ls1)  => sum_list_compute ls1 (h + n)
  end.
```

Definition of sum-list-compute which is the extension of sum-list:

```
Fixpoint sum_list_compute (ls1:list ) (n:nat){struct ls1}  : nat :=
  match ls1 with
    | nil  => n
    | (cons h ls1)  => sum_list_compute ls1 (h + n)
  end.
```

The definition of sum-list-correctness and its proof in Coq are shown below.

```
Lemma help: forall L x n, (x + n )+ sum_list' L = sum_list' (cons (x
+ n) L).
auto.
Qed.
Theorem sum_list_compute_relation:forall L x , sum_list'(cons x L)=
sum_list_compute L x.
induction L.
simpl.
auto.
simpl.
intros.
rewrite plus_assoc.
rewrite help.
rewrite IHL.
rewrite plus_comm.
auto.
Qed.
Theorem sum_list_correctness: forall L, sum_list L = sum_list' L.
induction L.
auto.
```

```
intros.
rewrite sum_list_compute_relation.
simpl.
auto.
Qed.
```

4.4 CASE STUDY: DEFINING LINES AND EXAMPLE PROOFS

In this section we first define a Point as in x-y coordinate system. Then we define a Line which can be constructed from two Points. Lines can be either parallel or intersecting. We axiomatically define this in all three theorem provers. If two lines are parallel, they cannot intersect or vice versa. We state base axioms to define this fact, and then state some conjectures to prove. In this case study, we also present proof by contradiction technique in all three theorem provers.

4.4.1 Definitions of Point and Line in Athena

We first declare symbols to represent points and lines in Athena first. We then define three functions which are parallel, intersect, and angle. A point can be formed by using two natural numbers. A line can be constructed by using two points. After declaring lines, we can state a function called parallel whose input is two lines and the output is a Boolean value. We repeat the same thing for the intersect function. If two lines are intersecting, this function will return true. It should return false otherwise. Here are the definitions in Athena:

```
(load-file "naturals.ath")
(domain Point)
(domain Line)
(declare point (-> (Nat Nat) Point))
(declare line (-> (Point Point) Line))
(declare parallel (-> (Line Line) Boolean))
(declare intersect (-> (Line Line) Boolean))
(declare angle (-> (Line Line) Nat))
```

We now define base axioms which will be used in the proof of a property which will be shown later. In the following we state in the first axiom that if two lines are parallel the angle between them is zero. The second axiom is saying that if two lines are intersecting, the angle between them cannot be zero. Here are the definitions:

```
(define parallel-axiom
   (forall ?l1 ?l2
      (if (parallel ?l1 ?l2)
          (= (angle ?l1 ?l2) zero))))

(define intersect-axiom
   (forall ?l1 ?l2
      (if (intersect ?l1 ?l2)
            (not (= (angle ?l1 ?l2) zero)))))

(assert parallel-axiom intersect-axiom)
```

In this proof, we use proof by contradiction. In inside the proof steps we also used modus ponnes “!mp” to prove some step. Here is the definition of parallel-not-intersect line property and proof:

```
 (define parallel-not-intersect
   (forall ?l1 ?l2
      (if (parallel ?l1 ?l2)
          (not (intersect ?l1 ?l2)))))
```

The proof of this property is done by using proof by contradiction technique. Here is the proof in Athena:

```
 (conclude parallel-not-intersect
  (pick-any l1 l2
    (assume (parallel l1 l2)
      (suppose-absurd (intersect l1 l2)
        (!absurd
          (conclude (= (angle l1 l2) zero)
            (!mp (!uspec* parallel-axiom [l1 l2])
                 (parallel l1 l2))
     (conclude (not (= (angle l1 l2) zero)
         (!mp (!uspec* intersect-axiom [l1 l2])
              (intersect l1 l2))))))))
```

4.4.2 Definitions of Point and Line in Isabelle

In Isabelle all definition and declaration are shown below. Here is the declaration of parallel, intersect and angle in Isabelle. Where parallel and intersection are functions of two lines which returns Boolean and angle is a function of two lines which returns natural number:

```
theory testing2 imports "~~/src/HOL/Main" begin
locale geometry =
  fixes parallel :: "'line ⇒ 'line ⇒ bool"
  and intersect :: "'line ⇒ 'line ⇒ bool"
  and angle :: "'line ⇒ 'line ⇒ nat"
  assumes axiom1: "parallel l1 l2 ⟹ angle l1 l2 = 0"
  and axiom2: "intersect l1 l2 ⟹ ¬ (angle l1 l2 = 0)"
begin
```

Here is the definition of parallel-not-intersect line property and proof:

```
lemma -- "parallel_not_intersect"
   assumes
          3: " parallel l1 l2"
```

```
  shows " ¬intersect l1 l2"
proof(rule ccontr)
  assume 4: "  ¬ ¬intersect l1 l2 "
have 5: " intersect l1 l2" using 4 by (simp)
    have 6: " angle l1 l2 = 0" using axiom1 3 by (simp)
    have 7: " ¬ (angle l1 l2 = 0)" using axiom2 5 by (simp)
  show False using 7 6 by (rule notE)
qed
end
```

4.4.3 Definitions of Point and Line in Coq

Here is the declaration of parallel, intersect and angle in Coq. Where parallel and intersection are functions of two lines which returns Boolean and angle is a function of two lines and return natural number:

```
Variable line: Set.
Variable parallel:line -> line -> bool.
Variable intersect: line -> line -> bool.
Variable angle : line -> line -> nat.
```

Here are the definitions of parallel line and intersect line axiom:

```
Hypothesis  axiom1: forall l1 l2:line,
 parallel l1 l2 = true -> angle l1 l2 =0.
Hypothesis axiom2 : forall l1 l2:line,
 intersect l1 l2 = true -> angle l1 l2 <>0  .
Hypothesis axiom3 : forall l1 l2:line,
 intersect l1 l2 = true.
```

Here is the definition of parallel-not-intersect line property and proof:

```
Lemma parrallel_not_intersect: forall l1 l2:line,

 parallel l1 l2 = true ->  intersect l1 l2 = false.

intros .

assert (angle l1 l2 = 0).

apply axiom1;apply H;assumption.

assert(angle l1 l2 <> 0).

apply axiom2;apply axiom3;assumption.

contradiction.

Qed.
```

CHAPTER FIVE

COMPARISON OF THEOREM PROVERS

5.1 JUSTIFICATION OF COMPARISON

5.1.1 Proof Readability and Understandability

Readability means something looking obviously clear to users. Based on the analysis carried out in proofs that are in different lengths, we found out that Athena proofs are usually longer than the proofs of coq because of its details are provided in the proof. The proof steps are written clearly in Athena which makes it more understandable for beginners of theorem provers. Isabelle proofs do not have too much details compared to Athena and Coq. From the case study of sum of n numbers, it is clear that Athena definitions and proofs are more readable and understandable without requiring answers for many questions while Isabelle and Coq definitions and proofs need more explanation about each line of the code written. The code in Isabelle and Coq looks more abstract for the beginners because the tactics need to be explained. The user has to learn tactics like unfold, rewrite, auto, syntax, terms and so on before writing a proof. An Example is given below for each theorem prover:

Athena Codes:

```
by-induction Equality{

  zero =>

       (!chain [(Factorial zero)

       -->(S zero)                             [Factorial-zero-axiom]

<--(FactorialEfficientHelper zero (S zero)) [FactorialEfficientHelper-zero-
axiom]

<--(FactorialEfficient zero)                [FactorialEfficient-axiom]])

| (S n) =>
```

```
     let {induction-hypothesis :=

             (Factorial n) = (FactorialEfficient n)}

        (!chain

        [(Factorial (S n))

-->((S n)*(Factorial n))                             [Factorial-succ-axiom]

-->((S n)*(FactorialEfficient n))                    [induction-hypothesis]

-->((S n)*(FactorialEfficientHelper n (S zero)))     [FactorialEfficient-axiom]

<--(FactorialEfficientHelper n ((S n)*(S zero)))     [P1]

<--(FactorialEfficientHelper (S n) (S zero))         [FactorialEfficientHelper-
succ-axiom]

<--(FactorialEfficient (S n))                     [FactorialEfficient-axiom]])}
```

Isabelle Code:

```
theorem Equality:"Fact n = Factorial_helper n 1"

apply(induct_tac n)

apply simp

apply simp

done
```

Coq code:

```
Lemma Equality_fact(n:nat):fact n = fn_fact n.

induction n.

rewrite fact_zero.

unfold fn_fact.

simpl.

auto.

unfold fn_fact.

rewrite fact_eq_strong.

simpl.

rewrite plus_0_r.

auto.

Qed.
```

In general shorter proofs are hard to understand while longer proofs present more proof steps which make them clearer.

5.1.2 Size of the Mathematical Libraries

The Isabelle system has a bigger library than both Coq and Athena systems. It is very difficult to measure the formal library size in an objective way [1].We used quantification methods to measure the size of all systems. The size of Isabelle system library is more than 50MB (including FOL, HOL and others theories in Isabelle) and Coq system library memory size is more than 5MB (including theories, Proofs, tactics and so on) while Athena is the smallest in size which is less than 2 megabytes. Theorem provers have theories. A theory is a collection of theorems and lemma in a formal library. Isabelle system library includes definitions and proofs for String, Real, Logics, Complex, Rational, Derivation in addition to first-order logics (FOL), ZF etc. Coq system library includes String, Real, Logics, Rational numbers etc. But Athena does not contain such mentioned theories in its library.

Table 5.1: Comparison of Theories in all the Systems

	Athena	Coq	Isabelle
Number of Theory	37	Approximately 240	More than 1000

Isabelle theorem prover has the bigger library as shown above, leading it to have more theorems inside its formal library.

5.1.3 Shorter Proofs

Table 5.2: Comparison of Theorem provers in terms of Lines

Theorem prover	Total number of lines
Athena	419Lines
Isabelle	126Lines
Coq	186Lines

A graph is plotted in figure 5.1 from the data in table 5.2 which shows clearly the differences of line of codes within the three theorem provers.

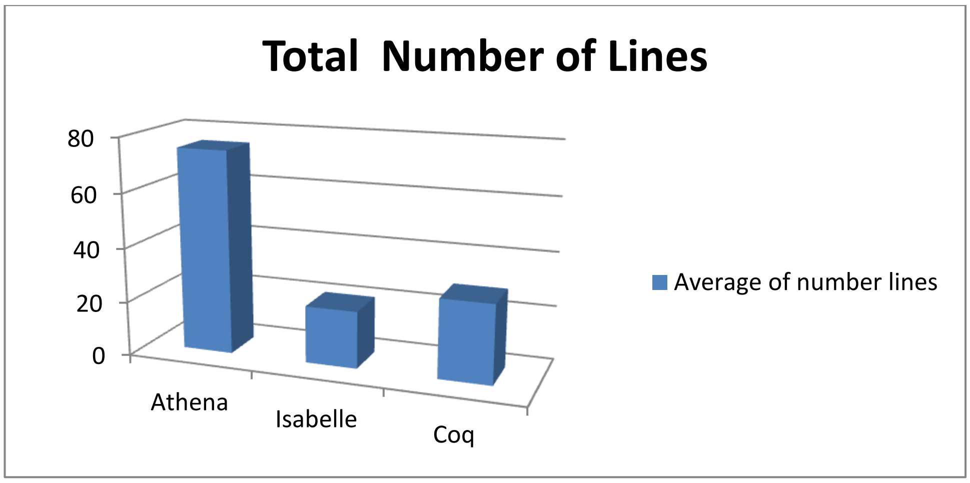

Figure 5.1: A graph showing the difference in terms of Lines

Most proofs in coq and Isabelle are solved automatically or semi-automatically which makes the proofs to skip many steps during the development. In general, Isabelle proofs are shorter compared to Coq and Athena proofs. In Coq, most steps are done either automatically or semi-automatically as in Isabelle. But Coq proofs consisted of a fewer lemmas and theorems in its formal library than Isabelle. Also, we found out that most of the proofs in Athena are almost doubling or tripling that of Isabelle and Coq in terms of lines of code as shown in table 5.2.

5.1.4 Flexibility in Constructing Proofs

Shorter proofs lead to less flexibility in constructing theorems. As opposed to this longer proofs make the theorem construction process more flexible since it is easier to move from one step to the next. This makes the user to direct a proof step to his desired direction and not directing the user to inconvenient steps.

Athena theorem prover is more flexible in constructing proofs compared to Isabelle and Coq theorem provers which use a simplifier and automatic tactics which can direct the user towards undesirable directions in some cases.

5.1.5 Better Documentation

Isabelle and Coq have well arranged documentation than Athena system. In Athena we have no specific site or link where we can find a good document that will be easy for us to understand Athena. There is only a brief guide published with only 13 pages. It is too brief to be of real use. Its link is:

http://proofcentral.org/athena/resources/understanding-athena-proofs.pdf

In terms of mailing lists for questions and contributions, Athena does not have any places where users can exchange ideas or send their questions. Nevertheless, we note that Athena's developer who is Kostas Arkoudas was very helpful whenever we had questions.

Coq and Isabelle systems have more advanced documentations. There are many tutorials and notes that explain their features. Coq has two main documents that are officially maintained by Coq development team which are Reference Manual and tutorials about the standard library distributed with the system. Also Coq has many published documents that might be used by both beginners and advanced users. These documents are:

- Coq'Art, Software Foundations,
- Coq in a Hurry (Yves Bertot, 2006),
- Video tutorials (Andrej Bauer, 2011) and many more (http://coq.inria.fr/documentation).

Also coq provides a website for questions and contributions via coq-club@pauillac.inria.fr.

Isabelle system also has reference manuals such as Isabelle (Paulson, 2002a), further logics of the system (FOL and ZF) and Isar (Nipkow et al., 2002). There is tutorial for Isabelle/HOL (Nipkow et al., 2002). All these are distributed freely with Isabelle at http://isabelle.in.tum.de/dist/

and it provides a website for questions and contributions via

cl-isabelle-users@list.cam.ac.uk. Finally a PhD thesis written by Wenzel which described the development of Isar in detail includes many reference applications and practical proof patterns but, dedicated tutorials and videos tutorials for beginners are still missing.

Table 5.3: Important Documents

S/N	**Document Type**	**Athena**	**Coq**	**Isabelle**
1.	Tutorial for beginners	yes	Yes	No
2.	Video tutorial	No	Yes	No
3.	Reference manual	No	Yes	Yes
4.	Mailing list	No	Yes	Yes

5.1.6 Availability on Platforms

In this case Isabelle, Athena and Coq systems are all available on different kind of platforms. The binaries of these systems are all free to download on the internet and could be used.

Isabelle and Athena systems are written in standard ML and can be run on all major platforms such as Intel, Apple, Sun and all major UNIX platforms. Coq system is written and implemented on Objective Caml (Ocaml), it runs on MS Windows, all Unix and Unix compatible systems, including MacOS X and Linux. Coq system binary is also free to download on internet.

Note that Athena does not have an integrated development environment which can make it easier for users to write proofs. It requires additional steps to install Athena before using it. Isabelle and Coq come with an IDE where users can start writing and executing proofs quickly.

5.1.7 Better Library Lookup

It is very difficult to remember the names of all definitions, lemmas and theorems available in the current context, especially if large libraries have been loaded. Also it is impossible to search for required lemmas or theorems manually by using brute force techniques. Athena does not have any mechanism to search for needed lemmas while, Isabelle and Coq systems provide tools to help users to find required lemmas.

Isabelle provides 'live' view of current theory context. Its `thms-containing` command finds theorem in a library: for example, `thms-containing` x < y x <= y

retrieves all facts involving the < and <= relations. Also you can use the find-textbox in Isabelle system to search for a required lemma by identifying some patterns: For example,

"(_ + _ = _)"

will retrieve all lemmas that has the structure shown above. This pattern specifically has a + sign and an equality sign. But the operators of + sign and the value on the right side of the equality sign are unknown. So any lemmas that have an addition operator as used above where '_' can be used in place of an arbitrary term will be returned.

Also Coq System provides some commands to lookup all the unknown lemmas or facts concerning predicates. These commands are `SearchAbout`, `SearchPattern` and `Search`. For example: `SearchAbout` < will retrieve all the relations about less than while, the `search` will only retrieve the lemmas where the searched predicate appears at the head position in the conclusion. `SearchPattern` allows finding the theorems with a conclusion matching a given pattern, where '_' can be used in place of an arbitrary term. Example: "`SearchPattern` (_ + _ = _)" .

5.1.8 More Feedback from the System

The Coq and Isabelle systems are fully interactive tactic-based provers. They are capable of showing out a lot of information at any line in code together with what to be proved, which is called goals or sub-goals. Athena lacks such vital information at each step rather someone needs to brainstorm to know where and what to do next.

Let us show this with an example in Isabelle and Coq systems to see how much information they provide, while Athena does not provide any help. The proof state in Isabelle at strong_fact where we apply (induct n) is:

```
proof (prove): step 1

goal (2 subgoals):

 1. ∀a. Factorial_helper 0 a = a * Fact 0

 2. ⋀n. ∀a. Factorial_helper n a = a * Fact n ⟹

        ∀a. Factorial_helper (Suc n) a = a * Fact (Suc n)
```

The proof state in Coq at fact_eq_strong where the induction step is proved is:

```
2 subgoal
______________________________________(1/2)
forall a : nat, helper_fact 0 a = a * fact 0
______________________________________(2/2)
forall a : nat, helper_fact (S n) a = a * fact (S n)
```

From above examples, we saw that Coq and Isabelle provided sub-goals which are base step and inductive step by itemizing them with subgoal1 and subgoal2 after applying induction tactic. From there you can know what to solve first and where to move forward. It is possible to implement functions that return the basis case and induction steps for a given property in Athena too. But these are not part of the language yet.

5.1.9 Less Meaningless Typographic Noise

Athena and Coq theorem prover texts contain less mathematically meaningless symbols than Isabelle theorem prover texts. In Isabelle theorem prover, there are some low-level “noise” characters that do not mean anything in mathematics. These characters are quotes (“”) around formulas at the input level. There are also (-), (- -) and dots (. And ..) . We show a related comparison in the next page:

Table 5.4: typographic Noise

S/n	Typographic Noise	Athena	Coq	Isabelle
1.	Quotes(””)	No	No	Yes
2.	?	Yes	No	Yes
3.	(-),(--)	No	No	Yes
4.	Single dot (‘.’)	No	Yes	Yes
5	Double dot (‘..’)	No	No	Yes

5.2 RESULT OF COMPARISONS

The table 5.4 shows the comparative result of the analysis conducted during this research. Having just one * means that the prover has a weakness for the item we used to compare. When the number of stars is more, the prover has more strength about the property we used to compare. There could be at most 4 stars in a row.

Table 5.4: Comparison of Systems.

		Athena	Isabelle	Coq
1	Readability and understandability	****	*	**
2	Bigger mathematical library	*	****	***
3	Shorter proofs	*	****	**
4	Flexibility in constructing proofs	****	*	**
5	Better documentation	*	**	****
6	Better library lookup	*	****	****
7	Available on more platforms	****	****	****
8	More feedback from the system	*	****	****
9	Less meaningless typographic noise	****	*	****

Note: For row-3 in the table 5.4, it is actually possible to write shorter proofs in Athena. The proof writer can write proofs whose parts can be proved by using theorem provers such as SPASS and Vampire etc. But these are external theorem provers.

CHAPTER SIX
CONCLUSIONS

We compared three theorem provers which are Athena, Coq and Isabelle based on some criteria as stated in our methodology section, where we pointed out some strong points in all the three systems. By representing some examples which are Equality of factorial and factorial efficient, Equality of Reverse and Reverse efficient, Sum of First n numbers, double reverse and followed by some properties necessary to be proved before completing the desired examples in all the three theorem provers.

Certainly, all systems have their inherent advantages and disadvantages. Speaking in terms of the system itself, the best choice for users is probably a matter of taste. In reality, the availability of existing background theory and proof tools is probably more important.

There are many similarities with the way we prove theorems in all these popular theorem prover. In some cases, it might be good to start with Athena, in other cases Coq and Isabelle might suit better. Since Isabelle's proofs are shorter, it can be easy to see the big picture or the steps of the proofs without getting into the details. If the problem domain requires one to see more details (such as solving equations in mathematics), starting with Athena might be better.

Concerning future work, it would be interesting to study how much of the advantages of Athena can be added to Isabelle and Coq, and vice versa. This Research may serve as a guideline for any such efforts towards better acceptance for development of proofs assistants by providing convincing computer assistance.

Some software can be implemented in a high-level language that shows all the three versions of the definitions and the proofs. With this way, one can write definitions and proofs in one language and can see it in other versions at the same time.

REFERENCES

[1] konstantine Arkodas, A case comaparison of Athena and HOL, June 7, 2001.

[2] Markus Wenzel and Freek Wiedijk, "A Comparison of The Mathematical Proof Languages: Mizar and Isar", *Journal of Automation Reasoning 29:389-411*, 2002

[3] Dana S. Scott, " The Seventeen Provers of the World" Freek Wiedijk (Eds), Springer; 2006 edition (February 3, 2006)

[4] Vicent Zammit, " A comparative Study of Coq and HOL",*10th international conference, TPHOLs,* 97 Murry Hill, NJ, USA, August 19-22, 1997, proceeding, PP 323-337, 1997.

[5] David Basin. Boyer-moore theorem and Nuprl proof: An experimental comparison. *In proceeding of first workshop on 'logical frameworks' Antibes, France,* pages 89-119. Cambridge university, press, 1991.

[6] Siekmann, J., Wrightson, G. (eds.*): Automation of Reasoning Classical Papers on Computational Logic 1957-1966*, vol. 1. Springer, Berlin (1983). http://www.intellektik.de/resources/OsnabrueckBuchfassung.pdf

[7] Davis, Martin (2001), "The Early History of Automated Deduction", in Robinson, Alan; Voronkov, Andrei,(Eds), *Handbook of Automated Reasoning* 1, *Pages 3-15*, Elsevier, 2001

[8] Bibel, Wolfgang (2007). *"Early History and Perspectives of Automated Deduction". KI 2007.* LNAI (Springer) (4667): 2–18. Retrieved 2 September 2012.

[9] H. Geuvers. "Proof Assistants:history, ideas and future*", in Sadhana Journal, Academy Proceedings in Engineering Sciences, Special Issue on Interactive Theorem Proving and Proof Checking, Indian Academy of Sciences*, Vol 34, part 1, February 2009, pp 3-25. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.5799&rep=rep1&type=pdf

[10] N. De Bruijn, Automath, a language for mathematics, Department of Mathematics, Eindhoven University of Technology, TH-report 68-WSK-05, 1968. Reprinted in revised form, with two pages commentary, in: *Automation and Reasoning, vol 2, Classical papers on computational logic 1967-1970*, Springer Verlag, 1983, pp. 159-200.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.5799&rep=rep1&type=pdf

[11] R.P. Nederpelt, H. Geuvers, R.C. de Vrijer, (editors), *Selected Papers on Automath, Volume 133 in Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam,* 1994, pp 1024.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.5799&rep=rep1&type=pdf

[12] P. Martin-Lof, *Intuitionistic type theory,* Napoli, Bibliopolis, 1984.

[13] B. Nordstrom, K. Petersson and J. Smith. *Programming in Martin-Lof 's Type Theory,* Oxford University Press, 1990.

[14] R.L Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki and S.F. Smith, *Implementing Mathematics with the Nuprl Development System, Prentice-Hall,* NJ, 1986.

[15] *Agda: An interactive proof editor*. http://agda.sf.net

[16] L. Magnusson and B. Nordstrom, *The ALF proof editor and its proof engine, In Types for Proofs and Programs,* eds. H. Barendregt and T. Nipkow, LNCS Vol 806, pp 213-237, 1994.

[17] http://en.wikipedia.org/wiki/Automated_theorem_proving#Comparison

[18] http://en.wikipedia.org/wiki/Proof_assistant

[19] Yegor Bryukhov, *Integration Of Decision Procedures Into High-Order Interactive Provers,* A Dissertation Submitted To The Graduate Faculty In Computer Science In Partial Fulfillment Of The Requirements For The Degree Of Doctor Of Philosophy, The City University Of New York,2006

[20] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation,* volume 78 of Lecture Notes in Computer Science. Springer-Verlag, NY, 1979.

[21] Richard B. Wells. *Mathematics and Mathematical Axioms,* 2006. http://www.mrc.uidaho.edu/~rwells/Critical%20Philosophy%20and%20Mind/Chapter%2023.pdf

[22] axioms. http://en.wikipedia.org/wiki/Axiom

[23] However, both theorems and theories are investigations. See Heath 1897 Introduction, The terminology of Archimedes, clxxxii:"theorem (θεώρνμα) from θεωρεῖν to investigate"

[24] theorem http://en.wikipedia.org/wiki/Theorem#Derivation_of_a_theorem

[25] Barwise, Jon,(ed.), *Handbook of Mathematical Logic*, Studies in Logic and the Foundations of Mathematics, Amsterdam, North Holland, 1982 ISBN 978-0-444-86388-1

[26] John N. Crossley: *What Is Mathematical Logic? A Survey*, 2011, http://www.csse.monash.edu.au/~jnc/jnc-tutorial.pdf

[27] James P. Bridge, *Machine learning and automated theorem proving,* November 2010, http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-792.pdf

[28] Stephen G. Simpson. *Logic and Mathematics*, 1999, http://www.personal.psu.edu/t20/papers/philmath/

[29] *Proof and other dilemmas: mathematics and philosophy* by Bonnie Gold, Roger A. Simons 2008 ISBN 0883855674 pages 12–20

[30] K. Arkoudas. *Athena,* 2006, http://proofcentral.org/athena.

[31] David R. Musser. *Understanding Athena Proofs.* Rensselaer Polytechnic Institute, Troy, NY 12180 http://proofcentral.org/athena/resources/understanding-athena-proofs.pdf.

[32] David R. Musser and Aytekin Vargun. *Proving Theorem with Athena.*

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.559&rep=rep1&type=pdf

[33] *What is Coq ? | The Coq Proof Assistant*. Coq.inria.fr. Retrieved on 2013-07-21

[34] PI.R2 http://www.pps.univ-paris-diderot.fr/pi.r2/

[35] Yves Bertot, *Coq in a Hurry*. April 2010, http://hal.inria.fr/docs/00/33/44/28/PDF/ coq- hurry.pdf

[36] Christine Paulin-Mohring, *Introduction to the Coq proof-assistant for practical software verification.* https://www.lri.fr/~paulin/LASER/course-notes.pdf

[37] Tinker, Miles A. (1963). *Legibility of Print*. Iowa: Iowa State University Press. pp. 5–7.ISBN 0-8138-2450-8.

[38] Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel, A Proof Assistant for Higher-Order Logic, December 5, 2013, http://isabelle.in.tum.de/doc/tutorial.pdf

[39] Lawrence C. Paulson: The foundation of a generic theorem prover. *Journal of Automated Reasoning,* Volume 5, Issue 3 (September 1989), Pages: 363-397, ISSN 0168-7433

[40] *Philosophical Papers, Volume 2* by Imre Lakatos, John Worrall, Gregory Currie, ISBN Philosophical Papers, Volume 2 by Imre Lakatos, John Worrall, Gregory Currie 1980ISBN 0521280303 pages 60–63

[41] *Evidence, proof, and facts: a book of sources*, Peter Murphy 2003 ISBN 0199261954, pages1–2

[42] *Logic in Theology – And Other Essays* by Isaac Taylor 2010 ISBN 1445530139 pages 5–15

[43] John Langshaw Austin: *How to Do Things With Words*. Cambridge (Mass.) 1962 – Paperback: Harvard University Press, 2nd edition, 2005, ISBN 0-674-41152-8.

[44] Cupillari, Antonella. The Nuts and Bolts of Proofs. Academic Press, 2001. Page 3.

[45] Alfred Tarski, Introduction to Logic and to the Methodology of the Deductive Sciences (ed. Jan Tarski). *4th Edition. Oxford Logic Guides,* No. 24. New York and Oxford: Oxford University Press, 1994, xxiv + 229 pp. ISBN 0-19-504472-X

[46] http://plato.stanford.edu/entries/justep-foundational/

[47] http://dictionary.reference.com/browse/proof

[48] Vargun, A. (2007, November). Termination checking without using an ordering relation. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications* (pp. 130-135). ACTA Press.

[49] Tobias Nipkow, Theorem Proving with Isabelle/HOL An Intensive Course, http://isabelle.in.tum.de/coursematerial/PSV2009-1/

[50] Adam Chlipala and George Necula, 2006, http://adam.chlipala.net/itp/lectures/sum.v

[51] Allen, R E, ed. (1984), *The Pocket Oxford Dictionary of Current English.* Oxford: Clarendon Press; p.421.

[52] Christine Paulin-Mohring, Tools for Practical Software Verification ,Lecture Notes in Computer Science Volume 7682, 2012, pp 45-95

[53] Chapter 1 Theorem proving with Coq, flint.cs.yale.edu/cs430/CoqTutorial.pdf

APPENDIX A

ATHENA PROOF

Appendix A consists of proofs pertaining Athena theorem provers which includes equality of reverse efficient and intuitive reverse proof and double reverse proof.

```
load "nat-plus.ath"

datatype (List T) := nil | (:: T (List T))

assert (datatype-axioms "List")

declare Join: (T) [(List T) (List T)] -> (List T)

 declare Reverse: (T) [(List T)] -> (List T)

 declare Reverse-helper: (T) [(List T) (List T)] -> (List T)

 declare Reverse-efficient: (T) [(List T)] -> (List T)


assert left-empty := (forall ?q . nil Join ?q = ?q)

  assert left-nonempty :=
   (forall ?x ?r ?q . (?x :: ?r) Join ?q = ?x :: (?r Join ?q))

  assert right-empty := (forall ?p . ?p Join nil = ?p)

  assert right-nonempty :=
   (forall ?p ?y ?q .
     ?p Join (?y :: ?q) = (?p Join (?y :: nil)) Join ?q)
 assert Associative :=
      (forall ?p ?q ?r .
      (?p Join ?q) Join ?r = ?p Join (?q Join ?r))

 assert empty := ((Reverse nil) = nil)

  assert nonempty :=
   (forall ?x ?r .
     (Reverse (?x :: ?r)) = (Reverse ?r) Join (?x :: nil))

assert  Reverse-helper-zero-axiom  :=  (forall  ?x  .  (Reverse-helper  nil  ?x)  =
?x)
assert Reverse-helper-succ-axiom :=
       (forall ?p ?r ?x . (Reverse-helper(?r :: ?p) ?x) =
                                 (Reverse-helper ?p (?r :: ?x)))
assert Reverse-Eff-axiom := (forall ?p . (Reverse-efficient ?p) =
                                             (Reverse-helper ?p nil))

define Reverse-correct :=  (forall ?p ?a .
                                     (Reverse-helper ?p ?a) =
                                  ((Reverse ?p) Join ?a))
by-induction Reverse-correct{
nil =>
pick-any a:(List 'T)
(!combine-equations
(!chain [(Reverse-helper nil a )
     --> a                       [Reverse-helper-zero-axiom]])
(!chain [((Reverse nil) Join a )
     -->   (nil Join a)                [empty]
     -->     a                         [left-empty]]))

| (x :: p) =>

let {induction-hypothesis := (forall ?a .  (Reverse-helper p ?a) =
                                        ((Reverse p) Join ?a))}
conclude (forall ?a .
          (Reverse-helper (x :: p) ?a) =
                            ((Reverse (x :: p)) Join ?a))
pick-any a:(List 'T)

(!combine-equations
(!chain [(Reverse-helper (x :: p) a)
       --> (Reverse-helper p (x :: a))   [ Reverse-helper-succ-axiom]
       --> ((Reverse p) Join (x :: a))   [induction-hypothesis]])
(!chain [((Reverse (x :: p)) Join a)
       --> (((Reverse p) Join (x :: nil)) Join a)     [nonempty]
       --> ((Reverse p) Join ((x :: nil) Join a))     [Associative]
       --> ((Reverse p) Join (x :: (nil Join a)))     [left-nonempty]
       --> ((Reverse p) Join (x :: a))                [left-empty]]))
}

define Equality-Reverse :=
   (forall ?n .
                ((Reverse-efficient ?n) = (Reverse ?n)))

by-induction Equality-Reverse  {
  nil =>
    (!combine-equations
    (!chain [(Reverse-efficient  nil)
         --> (Reverse-helper nil nil)         [Reverse-Eff-axiom]
         -->  nil                              [Reverse-helper-zero-axiom]])
    (!chain [(Reverse nil)
         --> nil                               [empty]]))

| (x :: p) =>
         let {induction-hypothesis := ((Reverse-efficient  p) = (Reverse p))}
     (!chain [(Reverse-efficient (x :: p))
              --> (Reverse-helper (x :: p) nil)        [Reverse-Eff-axiom]
            --> ((Reverse (x :: p)) Join nil)         [Reverse-correct]
            --> (Reverse (x :: p))                    [right-empty]])}
```

====================reverse-reverse=======================================

```
define join :=
   (forall ?p ?q .
     (Reverse (?p Join ?q)) = (Reverse ?q) Join (Reverse ?p))

  define reverse := (forall ?p . (Reverse (Reverse ?p)) = ?p)

  by-induction join {
    nil =>
    conclude (forall ?q . (Reverse (nil Join ?q)) =
                           (Reverse ?q) Join (Reverse nil))
      pick-any q
```

```
       (!combine-equations
         (!chain [(Reverse (nil Join q))
                  --> (Reverse q)            [left-empty]])
         (!chain [((Reverse q) Join (Reverse nil))
                  --> ((Reverse q) Join nil) [empty]
                  --> (Reverse q)            [right-empty]]))
| (x :: p) =>
  let {induction-hypothesis :=
         (forall ?q . (Reverse (p Join ?q)) =
                      (Reverse ?q) Join (Reverse p))}
  conclude (forall ?q . (Reverse ((x :: p) Join ?q)) =
                        (Reverse ?q) Join (Reverse (x :: p)))
    pick-any q
      (!chain [(Reverse ((x :: p) Join q))
                --> (Reverse (x :: (p Join q)))  [left-nonempty]
                --> ((Reverse (p Join q)) Join (x :: nil))
                                           [nonempty]
               --> (((Reverse q) Join (Reverse p)) Join (x :: nil))
                                           [induction-hypothesis]
               --> ((Reverse q) Join ((Reverse p) Join (x :: nil)))
                                           [Associative]
               <-- ((Reverse q) Join (Reverse (x :: p)))
                                           [nonempty]])
}

by-induction reverse {
  nil =>
  conclude ((Reverse (Reverse nil)) = nil)
    (!chain [(Reverse (Reverse nil))
             --> (Reverse nil)          [empty]
             --> nil                    [empty]])
| (x :: p) =>
  conclude ((Reverse (Reverse (x :: p))) = (x :: p))
    let {induction-hypothesis := ((Reverse (Reverse p)) = p)}
    (!chain
     [(Reverse (Reverse (x :: p)))
       --> (Reverse ((Reverse p) Join (x :: nil)))
                                                     [nonempty]
      --> ((Reverse (x :: nil)) Join (Reverse (Reverse p)))
                                           [join]
      --> ((Reverse (x :: nil)) Join p)    [induction-hypothesis]
       --> (((Reverse nil) Join (x :: nil)) Join p)
                                           [nonempty]
      --> ((nil Join (x :: nil)) Join p)   [empty]
      --> ((x :: nil) Join p)              [left-empty]
      --> (x :: (nil Join p))              [left-nonempty]
      --> (x :: p)                         [left-empty]])
}
```

APPENDIX B

COQ PROOF

Appendix A consists of proofs pertaining Coq theorem provers which includes equality of reverse efficient and intuitive reverse proof and double reverse proof.

```
Inductive list : Set :=
  | nil : list
  | cons : nat -> list -> list.

Fixpoint append (ls1 ls2 : list) {struct ls1} : list :=
  match ls1 with
    | nil => ls2
    | cons h t => cons h (append t ls2)
  end.

Fixpoint reverse (ls : list) : list :=
  match ls with
    | nil => nil
    | cons h t => append (reverse t) (cons h nil)
  end.

Fixpoint reverse'_helper (ls acc : list) {struct ls} : list :=
  match ls with
    | nil => acc
    | cons h t => reverse'_helper t (cons h acc)
  end.

Definition reverse' (ls : list) : list := reverse'_helper ls nil.

Theorem append_associative : forall ls1 ls2 ls3,
  append (append ls1 ls2) ls3 = append ls1 (append ls2 ls3).
  induction ls1; simpl; intuition.
  rewrite IHls1; trivial.
Qed.

Lemma reverse'_helper_correct : forall ls acc,
  reverse'_helper ls acc = append (reverse ls) acc.
  induction ls;simpl;intuition.
rewrite IHls.
  rewrite append_associative.
  trivial.
  Qed.

Lemma append_nil : forall ls, append ls nil = ls.
  induction ls; simpl; intuition.
  simpl.
auto.
congruence.
Qed.
```

```
Theorem reverse'_correct : forall ls, reverse' ls = reverse ls.
  intros.
  unfold reverse'.
  rewrite reverse'_helper_correct.
  apply append_nil.
Qed.
=====================reverse-reverse=======================================
Theorem reverse_app : forall l1 l2,
reverse (append l1 l2) = append (reverse l2) (reverse l1).
induction l1; simpl; intuition.
rewrite append_nil.
auto.
rewrite IHl1.
rewrite append_associative.
auto.
Qed.
Lemma Reverse_reverse : forall ls, reverse (reverse ls) = ls.
induction ls.
auto.
simpl.
rewrite reverse_app.
rewrite IHls.
simpl.
auto.
Qed.
```

APPENDIX C

ISABELLE PROOFS

Appendix A consists of proofs pertaining Isabelle theorem provers which includes equality of reverse efficient and intuitive reverse proof and double reverse proof.

```
theory Example1
imports Datatype
begin

datatype 'a lists = Nil
               | Cons 'a "'a lists"

primrec app :: "'a lists => 'a lists => 'a lists"
where
 "app Nil   L2       = L2"
| "app(Cons x  L1)  L2 =Cons x  (app L1  L2)"

 primrec reverse :: "'a lists => 'a lists" where

 "reverse Nil       = Nil"
| "reverse (Cons x  L1) = app (reverse L1)  (Cons x  Nil)"

primrec reverse_helper :: "'a lists ⇒'a lists ⇒ 'a lists" where
" reverse_helper Nil L2 = L2" |
" reverse_helper (Cons x L1) L2 =  reverse_helper L1 (Cons x L2)"

lemma app_Nil2 [simp] : "app L1  Nil = L1"
apply(induct_tac L1)
apply simp
apply simp
done

lemma app_assoc[simp] : "(app(app L1 L2)  L3) = (app L1  (app L2  L3))"
apply(induct_tac L1)
apply simp
apply simp
done

lemma strog_helper_Reverse[simp]  :"∀L2 .  reverse_helper L1 L2 = app (reverse
L1)  L2"
apply(induct_tac L1)
apply simp
apply simp
done
 theorem Reverse_Equality_Reverse_Eff:"  reverse_helper L Nil = reverse L"
 apply(induct_tac L)
 apply simp
 apply simp
 done
 =====================================reverse-reverse=====================
lemma rev_app[simp]: "reverse (app L1  L2) = app (reverse L2) (reverse L1)"
```

```
apply (induct L1)
apply (simp_all)
done

theorem reverse_reverse[simp]: "reverse (reverse L) = L"
apply (induct L)
apply (simp_all)
done
```