FORMALIZATION OF ASYMPTOTIC NOTATIONS IN HIGHER-ORDER-LOGIC

By

NADEEM IQBAL

2009-NUST-MS-M&S-06

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of Science Computational Science and Engineering

Research Centre for Modeling and Simulation,

National University of Sciences and Technology (NUST),

Islamabad, Pakistan.

(April 2012)

to my

LITTLE FAMILY

Acknowledgements

I owe an immense debt of gratitude to my advisor Dr. Osman Hasan whose supervision was a wonderful experience for me. My passions defy words for his continuous support, advice and encouragement throughout this project. His regularity, punctuality, prompt response of emails, his patience regarding my mistakes were all excellent. The way he guided me in understanding the intricacies and subtleties of the subject matter of my thesis in the limited time span, only the following *Rubai* by poet of East, Allama Iqbal, can help express my passions:

دم عارف نسیم صبح دم ہے
اسی سے ریشۂ معنی میں نم ہے
اگر کوئی شعیب آئے میسر
شبانی سے کلیمی دو قدم ہے

I express many thanks to Air Cdre Sikander Hayat Mirza, Principal RCMS whose emphasis for excellence kept me focused on my project. I thank him for his continuous support on different matters.

I would like to thank Dr. Khalid Pervez who guided me on the various aspects of my research thesis.

I present my respectful gratitude to Dr. Meraj Mustafa whose support during this thesis was very valuable.

I would also thank Dr. Aimal Rextin whose suggestions regarding different aspects of my research were of much significance.

I would like to thank Umair Siddique who gave me frequent help during the entire currency of my research thesis ranging from its conception to execution.

I would also like to thank Waseem Ahmad, Faiz Ali, Usman Sanwal and Binyameen Farooq for their support on different matters.

I will not forget to mention here MS Bushra Sadia who gave me valuable support regarding the use of Latex.

Finally, I am extremely gratified and indebted to all my family members especially my wife and children Rida Fatima, Uzair Nadeem and Irhah Fatima for their enormous support and patience.

Abstract

Asymptotic notations characterize the limiting behavior of a function. They are extensively used in many branches of mathematics and computer science particularly in analytical number theory, combinatorics and computational complexity while analyzing algorithms. Traditionally, the mathematical analysis involving these notations has been done by paper-and-pencil proof methods or simulation, which do not provide accurate results. Due to the enormous usage of computational algorithms in safety-critical domains these days, accuracy of asymptotic analysis has become extremely important. We propose to use Formal methods, which is a computer based mathematical analysis technique, to perform asymptotic analysis of algorithms. In order to introduce formal verification in this domain, this thesis provides the higher-order-logic formalizations of O, Θ, Ω, o and ω notations and the formal verification of most of their classical properties of interest. The formalization is based on the theory of sets, real and natural numbers and has been done using the HOL4 theorem prover. To demonstrate the practical usefulness and effectiveness of this formalization, we utilize it to formally analyze the computational complexities of the insertion sort and uniqueness of array elements algorithms. Our formalization bears immense potential for the formal analysis of cryptographic protocols in which the asymptotic notations are used to estimate the size of the key so that it will be infeasible to break a system using given number of steps.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Importance of Algorithms for Computing

Algorithms play a central role in Computing. Ranging from the human genome project to space exploration, from the nuclear cold test to online searching, from pollution tracking to oil and gas reservoir modeling, all these projects depend on the efficient algorithms which are running in the core of their implementations. We will discuss some important areas:

1.1.1 Genetics

The ultimate goal of human genome project [10] is to (i) identify all the 100,000 genes in human DNA (ii) determine the sequences of the 3 billion chemical base pairs that develop human DNA (iii) store this information in databases (iv) develop tools for data analysis. All the above listed steps necessitate sophisticated algorithms for their implementation. For example, Cropper and Anderson [11] used inverse analysis with two optimization techniques (Golden Section search and genetic algorithm (GA)) for the sake of determining fecundity and seedling demographic parameters which are in line with the observed secondary forest and mature distributions of forest size class. To realize this purpose, they used the genetic algorithm to the observed size class distributions in the age of 30 and for the mature forest (assumed ages of 65-150 years). Resultantly, the genetic algorithm provided a very good density-dependent models for each of the assumed mature forests ages.

1.1.2 Artificial Intelligence

The central theme of Artificial Intelligence (AI) is to mimic human reasoning with the help of algorithms. Without sophisticated algorithms, AI reduces to mere a philosophy. Today algorithms are being implemented in robotics [12], natural language processing [35], computer vision [16], expert systems [36] and automated reasoning [31] to name a few.

1.1.3 Cryptography

Cryptography is the science and art for developing different techniques for a secured communication in the presence of third parties, often called adversaries. In other words, we can say that in this discipline, various protocols are constructed and then analyzed for thwarting the influence of adversaries. These protocols include data integrity, data confidentiality, and authentication. Modern cryptography lies at the intersection of the disciplines of mathematics, computer science, and electrical engineering. For example, a fast, parallel and a secure cryptography algorithm using Lorenz's Attractor is presented in [32].

1.1.4 Linear Algebra

Linear algebra is central to modern mathematics and its applications. A traditional application of linear algebra is to find the solution of a system of linear equations in some unknowns. More advanced applications are in the fields of abstract algebra and functional analysis. Linear algebra is also used extensively in analytical geometry. It has a lot of applications in physics, computer sciences, engineering, natural sciences, and even in the social sciences. Furthermore, nonlinear mathematical models are usually approximated by their linear counterparts.

1.1.5 Data Routing on Internet

By using the Internet, people [10] round the globe can quickly access and retrieve large amounts of data. Behind all this activity, clever algorithms have been developed for the

efficient management and manipulation of large volumes of data. Particular algorithms developed for this purpose include finding the shortest path from the available routes on which the data will travel and to use a search engine to speedily access pages bearing a particular required information.

1.1.6 Operations Research

Operations research is the study about the optimal use of the resources. It explores different techniques of applying advanced analytical methods which would facilitate better decision-making. Mathematical modeling is such a technique for the analysis of complex situations. Operations research helps executives in making more effective decisions and building more productive systems. For instance, the usage of exact and heuristic algorithms is given in [13].

1.1.7 Electronic Commerce

Cyber revolution has affected all walks of life. Electronic commerce is no exception. Now the transactions of goods and services are done electronically [10]. The ability to secure necessary information such as credit card numbers, passwords are essential for this purpose. Digital signatures and public-key cryptography [39] algorithms are frequently used for this purpose.

1.2 Motivation

Traditionally, the computational time assessment of algorithms is done using benchmarking and asymptotic notations. In benchmarking, the main idea is to run the algorithms on a computer and then measure their speed in some time units. The benchmarking based assessment cannot be trusted completely because it measures the efficiency of a particular program which has been written in a specific language, running on a particular platform,

with a particular compiler and a particular input data [40]. From this single benchmark, it is difficult to predict that how much time the algorithm will take if it is deployed on a system with a different set of specifications. This limitation can be overcome by using an analytical approach based on asymptotic notations [10]. The first asymptotic notation, i.e., the Big-O notation or simply O-notation was introduced by a number theorist Bachmann in 1894 [5]. In the following years, its properties and physical interpretation became well understood, and it has been frequently used in algorithms analysis. Little-o notation was introduced by Landau in 1909 [29], and Big-Ω, Big-Θ, and Little-ω notations were first presented by Knuth in 1976 [28]. As with the O-notation, these asymptotic notations are all well understood and widely used in algorithm analysis [10].

The O-notation basically describes the worst-case scenarios and can be used to model the upper bounds on execution time required or space (memory) used by an algorithm depending on the given input size. Consider an algorithm that requires $8n^3 + 5n + 10$ execution steps. Now as the value of n increases, the contribution of the terms $5n$ and 10 in calculating the overall execution time of the algorithm gets smaller and smaller in comparison to $8n^3$ and thus they are ignored while computing the O-notation complexity of this algorithm. Similarly, the number 8 can also be discarded since it is not a "property" of the underlying algorithm. Thus, the running time of the algorithm is proportional to n^3 using the O-notation. The Θ-notation is used when a tighter bound than the one provided by O-notation over the complexity of an algorithm is required. For characterizing the lower bound on the execution time, the Ω-notation is used [10]. The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$, for instance, is asymptotically tight but the bound this $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. In the same fashion, we use ω-notation to denote a lower bound that is not asymptotically tight [10].

1.3 Algorithms Analysis Techniques

1.3.1 Paper-and-Pencil

The running time complexity analysis of algorithms has been traditionally done using paper-and-pencil proof methods. This technique does not guarantee the accuracy of the analysis results. It is prone to human error and there is always a risk of forgetting to pen down a critical assumption besides the proof. For example, in 1973, Hopcroft and Tarjan [22] proposed a *linear time algorithm* to decompose a graph into its tri-connected components, which is a very crucial algorithmic step for graph-theoretic problems that build upon it. However, their algorithmic description contains different flaws, which were discovered by Gutwenger and Mutzel [15] in 2001, when the highly complex algorithm was first implemented.

1.3.2 Simulation

Simulation [27], on the other hand, allows us to deduce the validity of a property by analyzing its behavior under a subset of all possible inputs. It is possible that our system is reported to be correct despite having some errors since the buggy inputs are not applied to the system. Thus, the analysis results are approximate due to the inherent incomplete nature of the simulation. Hence, both these traditional techniques of paper-and-pencil and simulation cannot be relied upon for the analysis of algorithms in safety critical domains such as medicine, transportation and avionics.

1.3.3 Formal Methods

In the past couple of decades, formal methods have been successfully used for the precise analysis of a verity of hardware and software systems. The rigorous exercise of developing a mathematical model for the given system and analyzing this model using mathematical reasoning usually increases the chances for catching subtle but critical design errors that are

often ignored by traditional techniques like paper-and-pencil based proofs or simulation. Given the extensive usage of asymptotic analysis of algorithms in safety-critical systems, there is a dire need of using formal methods support in this domain. However, due to the continuous nature of the analysis and the involvement of set theory principles, automatic state-based approaches, like model checking [25], cannot be used in this domain. On the other hand, we believe that higher-order-logic theorem proving [14, 19] offers a promising solution for conducting formal asymptotic analysis of algorithms. The main reason being the highly expressive nature of higher-order logic, which can be leveraged upon to essentially model any system that can be expressed in a closed mathematical form.

1.4 Asymptotic Notations

1.4.1 The O Notation

The O-notation provides an asymptotic upper bound for algorithms or functions. Its frequency of usage outnumbers the other notations because it gives us an upper bound on the complexity, i.e., it gives us a guarantee that at maximum a particular algorithm will take such a time for its execution. It can be defined for a given function g as follows:

Definition 1: *BigO Notation*

$$\vdash \forall\ \texttt{g.}\ \ \texttt{BigO (g:num} \rightarrow \texttt{real) = \{(f:num} \rightarrow \texttt{real)|}$$
$$\texttt{(}\exists\ \texttt{c n\_0.(}\forall\ \texttt{n.}\ \ \texttt{n\_0} \leq \texttt{n} \Longrightarrow \texttt{0} < \texttt{c} \wedge \texttt{0} \leq \texttt{f(n)} \leq \texttt{c} * \texttt{g(n)))\}$$

Here `f` and `g` are functions which take a natural number `num` and return a real number `real`. The constants `c` and `n_0` are of type `real` and `num`, respectively. The `BigO` takes a function g as an input and returns the set of all functions f which qualify the condition $\texttt{0} \leq \texttt{f(n)} \leq \texttt{c} * \texttt{g(n)}$. For example,

$$n^2 + 20n + 100 = O(n^2)$$

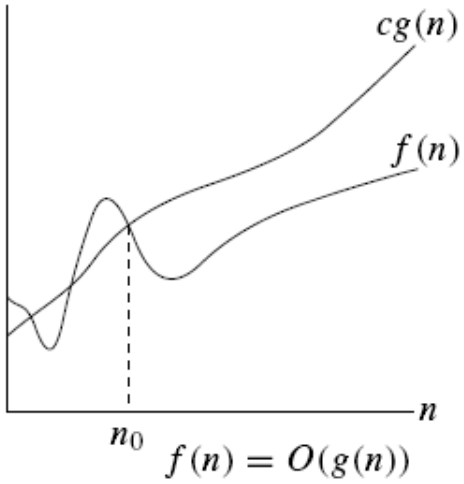

Figure 1.1: O-Notation

means that there exist some positive constants c and $n0$ such that $n^2 + 20n + 100 \leq cn^2$. It is to be noted that "=" is not used to express "is equal to" but it refers to set membership in the context of asymptotic notations. When the value of n is relatively small, functions $f(n)$ and $cg(n)$ behave haphazardly, but as soon as n crosses a certain threshold value $n0$, their behavior relative to each other becomes very smooth. Now, no matter how much large is the value of n, $f(n)$ can never cross $cg(n)$ as depicted by fig. 1.1.

O-notation is not a tight bound. It just provides an upper bound on the time that a particular algorithm can take for its execution. Heapsort, for instance, takes $O(n\ lgn)$ time for sorting n elements, since the call to BUILD_MAP_HEAP takes time $O(n)$ and each of the $n-1$ calls to MAX_HEAPIFY takes time $O(lgn)$.

1.4.2 The Θ Notation

The Θ-notation is used when we need a more strict asymptotic bound than the one provided by O. It is defined as follows:

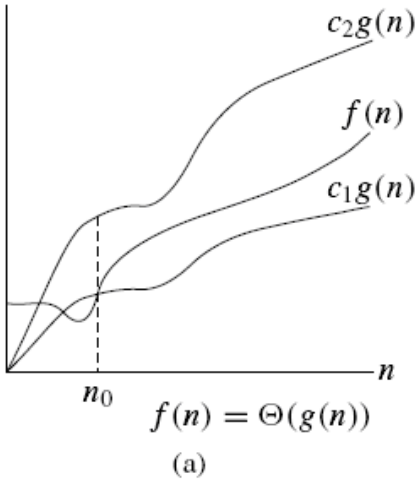

Figure 1.2: Θ-Notation

Definition 2: *BigTheta Notation*

$\vdash \forall$ `g.  BigTheta (g:num` $\rightarrow$ `real) = {(f:num` $\rightarrow$ `real)|`

`(`$\exists$ `c1 c2 n_0.(`$\forall$ `n.n_0` $\leq$ `n` $\Longrightarrow$ `0` $<$ `c1` $\wedge$ `0` $<$ `c2` $\wedge$

`0` $\leq$ `c1 * g(n)` $\leq$ `f(n)` $\leq$ `c2 * g(n)))}`

Here, `BigTheta` takes as an input a function g and returns the set of all functions f that satisfy a more strict condition, i.e.,`0` $\leq$ `c1` $*$ `g(n)` $\leq$ `f(n)` $\leq$ `c2` $*$ `g(n)` for all n greater than some $n0$ and for some real constants $c1$ and $c2$. For example,

$$n^2 + 20n + 100 = \Theta(n^2)$$

means that there exist some positive constants $c1$, $c2$ and $n0$ such that $c1n^2 \leq n^2+20n+100 \leq c2n^2$. Here again after certain value $n0$, $f(n)$ can never cross both its lower and upper bounds $c1g(n)$ and $c2g(n)$ respectively as demonstrated by fig. 1.2. The complexities of Merge sort and Counting sort are $\Theta(\ n\ lgn)$ and $\Theta(n)$, respectively.

1.4.3 The Omega Notation

The Ω-notation is used when a lower asymptotic bound is required. It is defined for a function g as follows:

Definition 3: *BigOmega Notation*

$\vdash \forall$ g. BigOmega (g:num $\rightarrow$ real) = {(f:num$\rightarrow$real)|
(∃ c n_0.(∀ n. n_0 $\leq$ n $\Longrightarrow$ 0 $<$ c $\wedge$ 0 $\leq$ c * g(n) $\leq$ f(n)))}

BigOmega also takes as an input function g and returns the set of all functions f which satisfy the condition $0 \leq c * g(n) \leq f(n)$. For example,

$$n^2 + 20n + 100 = \Omega(n^2)$$

means that there exist some positive constants c and $n0$ such that $cn^2 \leq n^2 + 20n + 100$. Ω-notation gives us the lower bound on the minimum time that an algorithm can take for its execution. After passing some particular value $n0$, $cg(n)$ can never cross $f(n)$ as shown in fig. 1.3. For example, the best case running time of sorting is $\Omega(n)$ when the array A is already sorted in the case of Insertion sort algorithm.

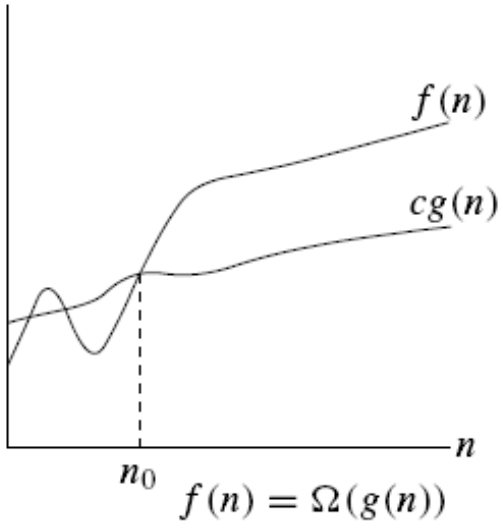

Figure 1.3: Ω-Notation

1.4.4 The LittleO Notation

The o-notation is used to denote an upper bound that is not asymptotically tight. It is defined for a function g as follows:

Definition 4: *LittleO Notation*

$\vdash \forall$ g. LittleO (g:num $\rightarrow$ real) = {(f:num $\rightarrow$ real)|
(∃ c n_0.(∀ n. n_0 ≤ n ⟹ 0 < c ∧ 0 ≤ f(n) < c * g(n)))}

The o-notation is a variant of BigO. It takes as an input a function and returns the set of all functions which qualify the condition $0 \leq f(n) < c * g(n)$ for all values of n greater than n_0 and for a given real constant c.

1.4.5 The LittleOmega Notation

The ω-notation is used to denote a lower bound that is not asymptotically tight. It is defined for a function g as follows:

Definition 5: *LittleOmega Notation*

$\vdash \forall$ g. LittleOmega (g:num $\rightarrow$ real) = {(f:num$\rightarrow$real)|
(∃ c n_0.(∀ n. n_0 ≤ n ⟹ 0 < c ∧ 0 ≤ c * g(n) < f(n)))}

LittleOmega is analogous to BigOmega. It takes as an input a function g and returns the set of all functions f which satisfy the condition $0 \leq c * g(n) < f(n)$ for all values of n greater than n_0 and for a given real constant c.

1.5 Applications of Asymptotic Notations

In this section, we will have a brief survey about the areas where the asymptotic notations are being extensively used.

1.5.1 Computational Complexity

The computational time of an algorithm is very important in this modern era. Traditionally, the analysis of algorithms has been done by benchmarking: running an algorithm on a machine and measuring its speed in some time units. There are a lot of parameters which influence the overall running time of an algorithm like hardware, compiler, operating system, programming language, input data etc. Due to these particularities, benchmarking can't provide us a sort of universal gauge for assessing the running time of an algorithm. We require such a setting that transcends all these limitations and that is the idea of asymptotic notations. Now the efficiency of algorithms is assessed by the theory of asymptotic notations [10].

1.5.2 Combinatorics

It is concerned with determining the total number of logical possibilities of some event without listing all the specific outcomes. One can often perform calculations involving probabilities simply by counting the possible outcomes. It often involves counting the number of possible permutations or combinations of a set of objects. When the number of elements becomes too large, the theory of asymptotic notations is used to characterize this explosion of arrangements.

1.5.3 Number Theory

Number theory is one of the oldest branches of mathematics. It is mainly concerned with the study of properties of natural numbers, rational numbers, irrational numbers, real numbers etc. The study of prime numbers is one of the main areas of number theory. Asymptotic notations, particulary BigO notation is extensively used to demonstrate different identities. For example,

$e^x = 1 + x + x^2/2 + O(x^3)$ as $x \to 0$

In the above identity, the most significant terms are written explicitly, then the least sig-

nificant terms are summarized by BigO notation.

The above identity expresses that the difference $e^x - (1 + x + x^2/2)$ is smaller than some constant times x^3 when x approaches to zero.

1.6 Related Work

The first higher-order-logic formalization of O-notation was presented by Avigad and Donnelly [4]. This formalization utilizes the *Ring Theory* to formalize O-notation and verify some of its classical properties using the Isabelle/HOL theorem prover. Ring theory is an abstract algebra that is not very well-known among computer scientists and thus using Avigad and Donnelly's seminal work for conducting formal asymptotic analysis is not a very straightforward task. In order to overcome this limitation, we extend Avigad and Donnelly's work by presenting a real-theoretic formalization of O-notation in this thesis. Moreover, we also present the formalization of a number of other asymptotic notations, i.e., Θ, Ω, o and ω [10]. Besides the formal definitions of these notations, the thesis also presents the formal verification of a very comprehensive set of their properties. The properties not only ensure the correctness of the reported definitions but also play a vital role in facilitating formal reasoning about asymptotic notation based complexity analysis of algorithms in the sound core of a higher-order-logic theorem prover. The main idea is to model the complexity of the algorithm in higher-order logic and then reason about its asymptotic bounds using our formalization of asymptotic notations using a theorem prover. This way, we can guarantee the accuracy of the analysis. For illustration purposes, we present the analyses of two algorithms: insertion sort and uniqueness of array elements in this thesis.

To the best of our knowledge, this is the first set-theoretic formalization of asymptotic notations in the open literature and thus allows us to utilize interactive theorem proving in a very interesting new direction. For illustration purposes, we present the analyses of two algorithms: insertion sort and uniqueness of array elements.

1.7 Proposed Methodology

The main theme of this thesis is to develop a theorem proving based framework for the formal analysis of the systems that involve the properties of asymptotic notations. Particularly, we developed a framework characterizing:

1. The ability to write the definitions of the asymptotic notations in HOL theorem prover that correspond to their mathematical counterparts so that we may in turn use them for our analysis. Owing to the fact that mathematical definitions involve integers, real numbers and set theory, so we used corresponding theories of our theorem prover.

2. The ability to formally verify the theorems and properties of the notations. These properties play a very important role for the formal analysis of the systems that involve asymptotic notations. Therefore, we proved a very comprehensive set of properties like transitivity, reflexivity, symmetry, transpose symmetry, sum, product, min etc. Of course, we used our previously written definitions of notations and a large number of theorems from the HOL theories during this process.

3. The ability to formally reason about the theorems, formalized in step 2, using a theorem prover. The proposed methodology, given in Figure 1.4, outlines the main idea behind the theorem proving based analysis of systems involving asymptotic notations. The grey shaded boxes in this figure represent the key contributions of the thesis that serve as basic requirements for conducting formal analysis of the systems involving asymptotic notations in a theorem prover. The input to this framework, depicted by two rectangles with curved bottoms, is the formal description of the system involving asymptotic notations that needs to be analyzed and a set of constraints (properties) that are required to be checked for the given system.

The first step in conducting formal analysis of the systems involving asymptotic notations is to write the time complexity of algorithms in higher-order-logic. For this purpose,

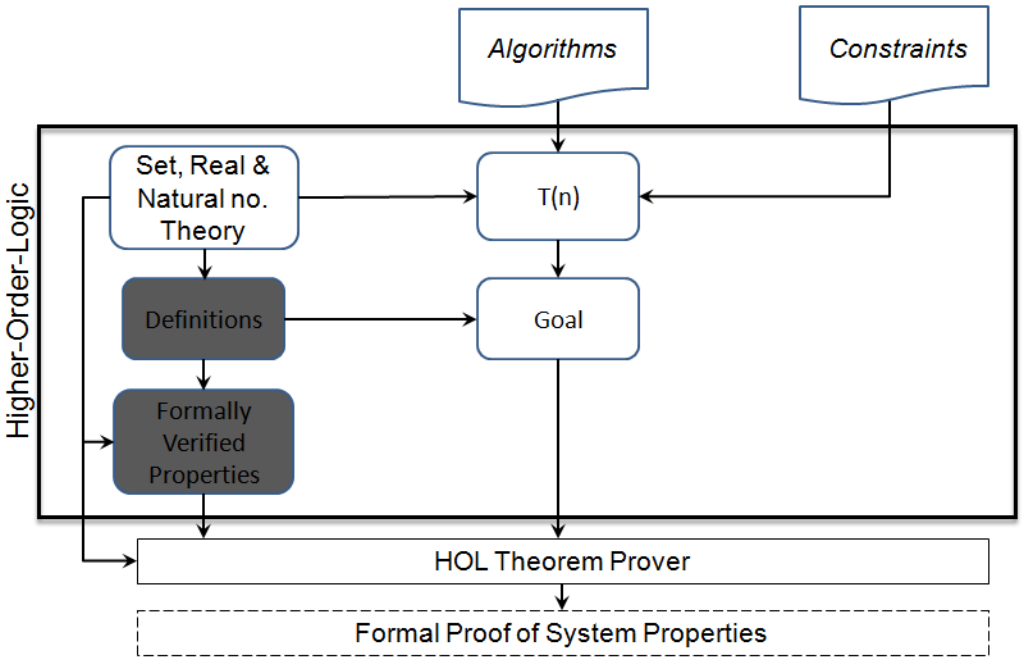

Figure 1.4: Proposed Methodology

we used set, real and natural number theories from theorem prover. The second step in the theorem proving based analysis of system involving asymptotic notations is to utilize the time complexity of algorithms, developed in the first step, to express system properties as higher-order logic theorems.

The third step for conducting analysis of system involving asymptotic notations in a theorem prover is to formally verify the higher-order-logic theorems developed in the previous step using a theorem prover. For this verification, it would be quite handy to have access to a library of some pre-verified theorems corresponding to some commonly used properties of the asymptotic notations like transitivity, reflexivity, symmetry, transpose symmetry, sum, product, max, min etc. Building on such a library of theorems would minimize the interactive verification efforts and thus speed up the verification process. Finally, the output of the theorem proving based framework of the system involving asymptotic notations, depicted by the rectangle with dashed edges, is the formal proofs of system properties that certify that the given system properties are valid for the given asymptotic notations based

system.

1.8 Thesis Contributions

The main goal of this thesis is the formal analysis of asymptotic notations using higher-order-logic theorem proving. With the help of this framework, we can do a very precise and accurate asymptotic analysis. The reported formalization can be used for the verification of a host of areas of algorithmics like cryptographic protocols, dynamic programming, sorting algorithms, string matching, greedy algorithms, number theoretic algorithms to name a few. More specifically, this thesis makes the following contributions:

1. It formalizes the definitions of Asymptotic notations like O, Θ, Ω, o and ω.

2. The above definitions provide a framework to formally verify a very comprehensive set of classical properties of interest like transitivity, reflexivity, symmetry and transpose symmetry etc.

3. It presents the formal modeling and verification of some benchmark algorithms like Insertion sort and uniqueness of array elements in higher-order-logic. The results we obtained are in complete harmony with [10, 30].

4. Cryptography protocols extensively use the asymptotic notations to estimate the size of the key so that it will be infeasible to break a system using given number of steps. Our framework can be used to formally verify the real-world properties used in these protocols. Moreover, our framework can also be used to formally verify the important properties of the above mentioned areas of algorithmics.

1.9 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 provides an overview of the HOL theorem prover which we have used for our formalization. It sheds light on the terminology which will be used by our thesis. Chapter 3 describes the formal definitions of asymptotic notations and then the formal verification of a comprehensive set of theorems and properties like transitivity, reflexivity, symmetry and transpose symmetry etc. To demonstrate the practical effectiveness and usefulness of our formalization infrastructure, Chapter 4 presents some applications like Insertion sort algorithm and uniqueness of an array. Lastly, chapter 5 concludes the thesis with some future directions of research.

Chapter 2

Preliminaries

In this chapter, we provide an introduction to the HOL theorem prover. The intent is to get some appreciation about the theories which we used and some notations [18].

2.1 Theorem Proving

Theorem proving [14, 19]: one of the most developed research areas in *automated reasoning*, is concerned with proving the mathematical theorems using a computer program. An array of different types of logics are used in this context like propositional logic, first-order logic, second-order logic or higher-order logic with increasing degree of expressibility. For example, the use of higher-order logic has many advantages over the first-order logic in terms of the availability of additional quantifiers at one's disposal. Higher-order logic is very rich in expressibility as compared to other logics. The underlying idea behind the theorem proving based formal analysis of a system is to mathematically construct the system in some appropriate logic and then various properties and theorems characterizing the system are verified by using computer based formal reasoning. Almost all theorem provers consists of two things: some axioms(facts) and primitive inference rules. The verification of the systems using theorem proving entails soundness as every new theorem must be created from the set of axioms and primitive inference rules and/or any other already proved theorems.

A theorem prover or a proof assistant is a tool which facilitates the formal description of a given system in terms of theorems(properties) and its proof through formal reasoning. Theorem provers have two types, i.e., automatic and interactive. In an interactive theorem

prover, as the name suggests, a significant user-computer interaction is required while automatic provers perform proof tasks automatically. A plethora of theorem provers exists but only a few of them are in continuous development and have large user community. Some commonly used automatic theorem provers include LeanTAP [7], Gandalf [43], METEOR [3], SETHEO [41], Otter [20] and MetiTarski [2]. While the interactive higher-order logic based theorem provers include Isabelle [44], Coq [8], HOL [42], HOL Light [21], ACL2 [26], PVS [37] and MIZAR [20].

This thesis uses the HOL theorem prover to conduct the formal analysis of the system involving asymptotic notations. The reason behind this selection is that it provides all the constructs, like built-in theories of sets, reals and integers, which are essential for the successful completion of the proposed thesis.

2.2 HOL Theorem Prover

HOL is an interactive theorem prover for the construction of mathematical theorems in higher-order logic and then proving them through formal reasoning. First version of HOL was developed by Mike Gordon at Cambridge University, in 1980′s. The core of HOL is interfaced to the functional programming language ML-Meta Language [38]. It utilizes the simple type theory of Church [9] with Hindley-Milner polymorphism [34] to implement higher-order logic. The first version of HOL is called HOL88 and other versions of HOL are HOL90, HOL98 and HOL4. HOL4, the latest version of HOL family, uses Moscow ML which is an implementation of Standard ML (SML). The core of HOL theorem prover consists of only 5 basic axioms and 8 primitive inference rules, which are implemented as ML functions. HOL has been extensively used for the formal verification of software and hardware systems. Apart from that, it has been successfully used for the formalization of pure mathematics.

2.2.1 Secure Theorem Proving

To ensure secure theorem proving, the logic in the HOL theorem proving system is represented in the strongly-typed functional programming language ML [38]. An ML abstract data type is used to represent higher-order logic theorems and the only way to interact with the theorem prover is by executing ML procedures that operate on values of these data types. Soundness is guaranteed at every step as every new theorem must be verified by applying these basic axioms and primitive inference rules and/or any other previously verified theorem(s).

2.2.2 Terms

HOL has four types of terms: constants, variables, function applications, and lambda-terms. Variables are strings of letters or digits beginning with a letter, e.g., x, c, num3 etc. The syntax of the constants is similar to that of variables, but they cannot be bounded by quantifiers. The type of an identifier, i.e., variable or a constant, is determined by a theory; e.g., F, T. Applications in HOL represent the evaluation of a function f at an argument x. We can also use λ-terms, also called lambda abstractions for the sake of denoting functions. λ-terms has the syntax $\lambda x.f(x)$ and represents a function which takes x as an argument and returns $f(x)$ as the function value.

2.2.3 Inference Rules

Inference rules are well-defined procedures for deriving new theorems from axioms and/or already proved theorems and they are represented as ML functions. There are eight primitive inference rules in HOL. These rules are *Assumption introduction*, *Reflexivity*, *Beta-conversion*, *Substitution*, *Abstraction*, *Type instantiation*, *Discharging an assumption* and *Modus Ponens* [1].

2.2.4 Theorems

A theorem is a statement written in higher-order-logic that may be an axiom and/or follows from theorems by an inference rule. A theorem consists of a finite set of boolean terms Ω called the assumptions and a boolean term S called the conclusion. For example, if (Ω, S) is a theorem in HOL then it is written as $\Omega \vdash S$.

2.2.5 Theories

A HOL theory consists of definitions, axioms, theorems, a set of types, type operators, constants. It consists of a list of theorems that have already been proved from the basic axioms and definitions. A user usually loads the required theories to use the definitions and theorems which are part of that theory. The availability of HOL theories facilitates the user to utilize the existing theorems without re-inventing the wheel and extending the theories when required. HOL theories are organized in a hierarchical fashion and theories can have other theories as parents and all of the types, constants, definitions, axioms and theorems of parent theory can be used in the child theory. For example, one of the basic theories in HOL is `bool` and this is also parent theory of `operator`. We utilized the HOL theories of real numbers, natural numbers and sets in our work. One of the primary reasons for selecting the HOL theorem prover for our work was to get benefit from these built-in mathematical theories.

2.2.6 Writing Proofs in HOL

In HOL, there are two types of interactive proof methods: forward and backward. In a forward proof method, a user starts from the primitive inference rules and proves the goals on top of these rules and already proved theorems. The forward proof method is a very difficult approach as it requires all the low level details of the proof in advance. A backward or a goal directed proof method is the reverse of the forward proof method. It is based on the concept of a tactic, which is an ML function that breaks goals into simpler

sub-goals. In the goal directed proof method, the user starts with the desired theorem or the main goal. The main goal is further reduced to simpler subgoals using the *tactics* which are ML-functions. There are many automatic proof procedures and proof assistants [17] available in HOL which help the user in directing the proof till the job is done. In interactive theorem proving, the user interacts with HOL proof editor and guides the prover using the necessary tactics until the last step of the proof is reached. In HOL, some of the proof steps are solved automatically while others require significant user interaction.

2.2.7 HOL Notations

Table 2.1 provides the mathematical interpretations of some frequently used HOL symbols and functions in this thesis.

Table 2.1: HOL Symbols and Functions

HOL Symbol	**Standard Symbol**	**Meaning**
/\	and	Logical *and*
\/	or	Logical *or*
~	not	Logical *negation*
==>	$\longrightarrow$	Implication
<==>	$=$	Equality
!x.t	$\forall x.t$	for all $x : t$
?x.t	$\exists x.t$	for some $x : t$
λx.t	$\lambda x.t$	Function that maps x to $t(x)$
num	$\{0, 1, 2, \ldots\}$	Positive Integers data type
real	All Real numbers	Real data type
suc n	$(n + 1)$	Successor of natural number
ln x	$log_e(x)$	Natural logarithm function
abs x	$\|x\|$	Absolute function
min x y	$min(x, y)$	Minimum of x and y
max x y	$max(x, y)$	Maximum of x and y
FACT n	$n!$	Factorial of n
inv x	$1/x$	Inverse of x

Chapter 3

HOL Formalization of Asymptotic Notations

In this section, we define and prove some of the key properties of asymptotic notations (O, Θ, Ω, o and ω), such as transitivity, product and transpose symmetry. The formal verification of these properties not only ensures the correctness of our formal definitions but also paves the way to the formal reasoning about the complexity of real-world algorithms and protocols.

3.1 The O Notation

We will discuss at length the formal proof details of a couple of key theorems of O-notation here:

Theorem 1: *Transitivity of O-Notation*

$\vdash \forall$ f g h. f $\in$ (BigO g) $\wedge$ g $\in$ (BigO h) $\Longrightarrow$ f $\in$ (BigO h)

Proof Sketch: We start the proof process by rewriting the above goal with the definition of BigO notation (Definition 1) along with some set-theoretic simplifications and we reach the following subgoal:

$$
\begin{aligned}
&(\forall \text{ n. } \text{n\_0} \leq \text{n} \Longrightarrow 0 < \text{c} \wedge 0 \leq \text{f(n)} \wedge \text{f(n)} \leq \text{c} * \text{g(n)}) \wedge \\
&(\forall \text{ n. } \text{n\_0'} \leq \text{n} \Longrightarrow 0 < \text{c'} \wedge 0 \leq \text{g(n)} \wedge \text{g(n)} \leq \text{c'} * \text{h(n)}) \\
&\qquad\qquad \Longrightarrow (\exists \text{ c n\_0}.\forall \text{ n. } \text{n\_0} \leq \text{n} \Longrightarrow 0 < \text{c} \wedge \\
&\qquad\qquad\qquad\qquad 0 \leq \text{f(n)} \wedge \text{f(n)} \leq \text{c} * \text{h(n)})
\end{aligned}
$$

At this stage, we have to provide specific values for the variables c and n_0. We specialized c and n_0 with $c * c'$ and MAX(n_0 n_0′), respectively. From the assumptions $0 < c$ and $0 < c'$, we can readily deduce that $0 < c * c'$. Furthermore, the function MAX returns maximum of the given pair of natural numbers. To fulfill the requirements in both the assumptions, we require such a value of n_0 that would fulfil both these conditions and that value is MAX(n_0 n_0′). Obviously, when the maximum of these two numbers will be less n, the other number will be automatically less than n too. By some straightforward arithmetic reasoning, we formally verified the given subgoal.

Theorem 2: *Sum of O-Notation*

$$\vdash \forall\ t1\ t2\ g1\ g2.\ \ t1 \in (BigO\ g1) \wedge t2 \in (BigO\ g2)$$
$$\Longrightarrow (\lambda n.\ \ t1\ n + t2\ n) \in (BigO\ (max(g1, g2)))$$

Proof Sketch: We start the proof process by rewriting the above goal with the definition of BigO notation (Definition 1) along with some set-theoretic simplifications and we reach the following subgoal:

$$(\forall\ n.\ \ n\_0 \leq n \Longrightarrow 0 < c \wedge 0 \leq t1(n) \wedge t1(n) \leq c * g1(n)) \wedge$$
$$(\forall\ n.\ \ n\_0' \leq n \Longrightarrow 0 < c' \wedge 0 \leq t2(n) \wedge t2(n) \leq c' * g2(n))$$
$$\Longrightarrow (\ \exists\ c\ n\_0.\forall\ n.\ \ n\_0 \leq n \Longrightarrow 0 < c \wedge 0 \leq (t1(n) + t2(n)) \wedge$$
$$(t1(n) + t2(n)) \leq c * max(g1(n), g2(n)))$$

Here for fulfilling the conditions of the assumptions $n\_0 \leq n$ and $n\_0' \leq n$, we again specialized n_0 by MAX(n_0, n_0′). Furthermore, we specialized c with $2 * max(c,\ c')$. With some straightforward arithmetic reasoning, we proved our goal.

Moreover, we formally verified the following theorems of *O*-notation in HOL4 and the formal reasoning details can be found in our proof script [24].

1. $\vdash \forall\ t1\ t2\ g1\ g2.\ \ t1 \in (BigO\ g1) \wedge t2 \in (BigO\ g2)$
$\Longrightarrow (\lambda n.\ \ t1\ n * t2\ n) \in (BigO\ (g1 * g2))$

2. $\vdash \forall$ f g. f $\in$ (BigO g) $\Longrightarrow \forall$ k. (λn. k * f n) $\in$ (BigO g)

3. $\vdash \forall$ f. f $\in$ (BigO f)

4. $\vdash \forall$ f g. f $\in$ (BigO g) $\Longrightarrow$ g $\in$ (BigOmega f)

5. $\vdash \forall$ f g. f $\in$ (BigO g) $\wedge$ f $\in$ (BigOmega g)
$\Longrightarrow$ f $\in$ (BigTheta g)

6. $\vdash \forall$ n. FACT n $\in$ (BigO (FACT (n + 1))

The first property in the above list is called the product property. If an algorithm having complexity t1 is run t2 times or vice versa, then the product of their complexities will still lie in the BigO of their product of orders of growth. The second property refers to a situation in which if an algorithm is run for k times, its overall complexity in this case will still lie in the same order of growth. The third property is a very interesting property of O-notation and is called reflexivity. It states that the complexity of an algorithm is always its own order of growth.

3.2 The Θ Notation

This subsection describes the formally verified properties of the Θ-notation using HOL4.

Theorem 3:

$\vdash \forall$ f g. f $\in$ (BigTheta g) $\Longleftrightarrow$ g $\in$ (BigTheta f)

Theorem 3 is the symmetry property of Θ-notation. We proceed with its verification by splitting the main goal into the following two subgoals:

$\vdash \forall$ f g. f $\in$ (BigTheta g) $\Longrightarrow$ g $\in$ (BigTheta f)

$\vdash \forall$ f g. g $\in$ (BigTheta f) $\Longrightarrow$ f $\in$ (BigTheta g)

The proof sketch for the first subgoal is given below and the other one was handled in a very similar way.

Proof Sketch: We start the proof process by rewriting the goal with the definition of BigTheta notation (Definition 2) along with some set-theoretic simplifications and we reach the following subgoal:

$$
\begin{aligned}
&(\forall\ \texttt{n}.\ \ \texttt{n\_0} \leq \texttt{n} \Rightarrow 0 < \texttt{c1} \wedge 0 < \texttt{c2} \wedge 0 \leq \texttt{c1} * \texttt{g(n)} \wedge \\
&\quad \texttt{c1} * \texttt{g(n)} \leq \texttt{f(n)} \wedge \texttt{f(n)} \leq \texttt{c2} * \texttt{g(n)}) \\
&\qquad \Longrightarrow (\exists\ \texttt{c1}\ \texttt{c2}\ \texttt{n\_0}.(\forall\ \texttt{n}.\ \ \texttt{n\_0} \leq \texttt{n}) \Rightarrow 0 < \texttt{c1} \wedge 0 < \texttt{c2} \wedge \\
&\qquad\qquad 0 \leq \texttt{c1} * \texttt{f(n)} \wedge \texttt{c1} * \texttt{f(n)} \leq \texttt{g(n)} \wedge \texttt{g(n)} \leq \texttt{c2} * \texttt{f(n)})
\end{aligned}
$$

Here again, the formal reasoning process relies on choosing the right set of values of variables `n_0`, `c1` and `c2`. We chose them to be `n_0`, `1/c2` and `1/c1`, respectively, and verified the subgoal based on arithmetic reasoning.

Moreover, we proved the following theorems of Θ-notation [24]:

1. $\vdash \forall\ \texttt{f}\ \texttt{g}\ \texttt{h}.\ \ \texttt{f} \in \texttt{(BigTheta g)} \wedge \texttt{g} \in \texttt{(BigTheta h)}$
   $\Longrightarrow \texttt{f} \in \texttt{(BigTheta h)}$

2. $\vdash \forall\ \texttt{t1}\ \texttt{t2}\ \texttt{g1}\ \texttt{g2}.\ \ \texttt{t1} \in \texttt{(BigTheta g1)} \wedge \texttt{t2} \in \texttt{(BigTheta g2)}$
   $\Longrightarrow (\lambda\texttt{n}.\ \ \texttt{t1 n + t2 n}) \in \texttt{(BigTheta (g1 + g2))}$

3. $\vdash \forall\ \texttt{f}.\ \ \texttt{f} \in \texttt{(BigTheta f)}$

4. $\vdash \forall\ \texttt{f}\ \texttt{g}.\ \ \texttt{f} \in \texttt{(BigTheta g)} \Longrightarrow \texttt{g} \in \texttt{(BigTheta f)}$

The first property is the transitivity of BigTheta. If complexity of some algorithm, say `f` lies in BigTheta of some another function `g` and complexity `g` of the function in turn, lies in BigTheta of yet another function `h`, then according to this property, `f` will lie in BigTheta of `h`. The second property is called the sum property of BigTheta. If complexities of two algorithms lie in BigTheta, then the sum of their complexities will also lie in the

corresponding sum of the orders of growth in the BigTheta. The third property in the above list is called the reflexivity of BigTheta.

3.3 The Ω Notation

This subsection describes the formal verification of some classical properties of the Ω-notation.

Theorem 4:

$$\vdash \forall\ \texttt{t1 t2 g1 g2.}\ \ \texttt{t1} \in \texttt{(BigOmega g1)} \wedge \texttt{t2} \in \texttt{(BigOmega g2)}$$
$$\Longrightarrow (\lambda\texttt{n.}\ \ \texttt{t1 n + t2 n}) \in \texttt{BigOmega (min (g1, g2))}$$

Theorem 4 implies that if an algorithm consists of two parallel components then the algorithm's overall efficiency will be determined by the part with a lower order of growth.

Proof Sketch: We start the proof process by rewriting the goal with the definition of BigOmega notation (Definition 3) along with some set-theoretic simplifications and we reached to the following goal:

$$(\forall\ \texttt{n.}\ \ \texttt{n\_0} \leq \texttt{n} \Rightarrow 0 < \texttt{c} \wedge 0 \leq \texttt{c} * \texttt{g1(n)} \wedge \texttt{c} * \texttt{g1(n)} \leq \texttt{t1(n)}) \wedge$$
$$(\forall\ \texttt{n.}\ \ \texttt{n\_0'} \leq \texttt{n} \Rightarrow 0 < \texttt{c'} \wedge 0 \leq \texttt{c'} * \texttt{g2(n)} \wedge \texttt{c'} * \texttt{g2(n)} \leq \texttt{t2(n)})$$
$$\Longrightarrow (\ \exists\ \texttt{c n\_0.}\forall\ \texttt{n.}\ \ \texttt{n\_0} \leq \texttt{n} \Rightarrow 0 < \texttt{c} \wedge 0 \leq \texttt{c} * \texttt{min(g1(n), g2(n))}$$
$$\wedge\ \texttt{c} * \texttt{min(g1(n), g2(n))} \leq \texttt{(t1(n) + t2(n))}$$

Then we specialized the variables c and n_0 with $2 * \texttt{min}(\texttt{c}, \texttt{c}')$ and $\texttt{MAX}(\texttt{n\_0},\ \texttt{n\_0}')$, respectively. This specialization allowed us to discharge all the conditions of the two assumptions and thus in turn prove the above subgoal using some arithmetic reasoning.

Theorem 5:

$$\vdash \forall\ \texttt{n.}\ \ \texttt{FACT(n+1)} \in \texttt{BigOmega (FACT (n))}$$

Proof Sketch: We start the proof process by rewriting above goal with the definition of BigOmega notation (Definition 3) and the HOL4 factorial function FACT along with some set-theoretic simplifications. Then we applied induction on `n` and after some arithmetic reasoning, we proved our goal.

Moreover, we proved the following theorems of Ω-notation [24]:

1. $\vdash \forall$ `f g h.  f` $\in$ `(BigOmega g)` $\wedge$ `g` $\in$ `(BigOmega h)`
   $\Longrightarrow$ `f` $\in$ `(BigOmega h)`

2. $\vdash \forall$ `f g.  f` $\in$ `(BigO g)` $\Longrightarrow$ `g` $\in$ `(BigOmega f)`

3. $\vdash \forall$ `f.  f` $\in$ `(BigOmega f)`

3.4 The LittleO Notation

We verified the transitivity and transpose symmetry properties of the LittleO notation as the following higher-order-logic theorems [24].

1. $\vdash \forall$ `f g h.  f` $\in$ `(LittleO g)` $\wedge$ `g` $\in$ `(LittleO h)`
   $\Longrightarrow$ `f` $\in$ `(LittleO h)`

2. $\vdash \forall$ `f g.  f` $\in$ `(LittleO g)` $\Longleftrightarrow$ `g` $\in$ `(LittleOmega f)`

3.5 The LittleOmega Notation

We verified the following properties of the LittleOmega notation as the following higher-order-logic theorems [24].

1. $\vdash \forall$ `f g.  f` $\in$ `(LittleOmega g)` $\wedge$ `g` $\in$ `(LittleOmega h)`
   $\Longrightarrow$ `f` $\in$ `(LittleOmega h)`

2. $\vdash \forall$ `f g.  f` $\in$ `(LittleOmega g)` $\Longleftrightarrow$ `g` $\in$ `(LittleO f)`

The formalization reported in this thesis so far took about 300 man-hours and about 1200 lines of HOL4 proof script, which is available for download at [24]. The main challenging part in the verification process was to pick the right values for the existentially quantified variables as has been described in the proof sketches of some of the theorems above. The rest of the verification process was based on arithmetic and set-theoretic simplification, which was greatly aided by the automatic verifiers and simplifiers along with the rich library of pre-verified theorems for set, natural and real number theories available in HOL4.

We tried to verify a comprehensive set of properties of the considered five asymptotic notations. The formal verifications of these properties, which are available in textbooks on asymptotic notations [10, 30], guarantee our formal definitions to be correct. Moreover, the formal verification of these properties is expected to facilitate the process of formal reasoning about asymptotic notations as will be demonstrated in the next chapter of this thesis.

Chapter 4

Applications

In this section, we will present the formal asymptotic analysis of two algorithms namely insertion sort and uniqueness of array elements.

4.1 Insertion Sort Algorithm

Sorting is a process of listing the items in some logical order. The order may be ascending, descending, alphabetical, chronological or even topological. Sorting lies at the heart of many computer applications involving number crunching, searching, information retrieval and data mining. A plethora of sorting algorithms exists like insertion sort, mergesort, quicksort, heapsort, counting sort and radix sort [10]. Every sorting algorithm has its pros and cons and usually, the one which best suits the situation is deployed to the real world problems. Here the main yardstick for comparing the efficiency of competing algorithms for implementation is the time and space complexity which are analyzed using the asymptotic notations. Formally, *time complexity* of an algorithm is the total number of basic operations performed, expressed as a function of the input size. Similarly, *Space complexity* is the amount of storage needed to solve the problem, typically expressed as a function of the input size (number of bits to represent input).

Insertion sort [10] is a simple sorting algorithm that is more efficient for small lists and often is used as a part of more sophisticated sorting algorithms. Insertion sort algorithm works by picking elements from the given list one by one and inserting them in their correct position into a new sorted list. Algorithm 1 describes the main computational steps of an insertion sort algorithm along with the cost and time steps required to execute each

statement. Here, t_j represents the number of times the *while loop* runs for the value j and constants c_i represent the cost of the ith statement. The general case time complexity ($T(n)$) of Insertion sort algorithm is given as following:

$$T(n) = c_1 n + (c_2 + c_4 + c_7)(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) \qquad (4.1)$$

Algorithm 1 Insertion Sort Algorithm	`Cost`	`TimeSteps`
procedure InsertionSort(A)		
for $j = 2 \leftarrow n$ **do**	▷ `c1`	`n`
$key \leftarrow A[j]$	▷ `c2`	$\mathtt{n}-1$
$i \leftarrow j - 1$	▷ `c3`	$\mathtt{n}-1$
while $i > 0$ `and` $A[i] > key$ **do**	▷ `c4`	$\sum_{\mathtt{j}=2}^{\mathtt{n}} \mathtt{t_j}$
$A[i+1] = A[i]$	▷ `c5`	$\sum_{\mathtt{j}=2}^{\mathtt{n}} (\mathtt{t_j} - 1)$
$i \leftarrow i - 1$	▷ `c6`	$\sum_{\mathtt{j}=2}^{\mathtt{n}} (\mathtt{t_j} - 1)$
end while		
$A[i+1] \leftarrow key$	▷ `c7`	$\mathtt{n}-1$
end for		
end procedure		

The analysis of insertion sort algorithm involves two main cases, i.e., *best case* and *worst case*: In the best case, $t_j = 1$ and the control will never enter into the body of the loop and only *while loop* header will contribute to the cost. While, the in worst case $tj = j$, i.e., the control will enter into the body of the *while loop* for each iteration of the *for loop* and the *while loop* will iterate at its maximum.

Now, we will present the formal verification of the asymptotic bounds of the running time of the insertion sort algorithm, for the best case and the worst case. The first step in the formal verification of the asymptotic bounds of the running time of Insertion sort algorithms is to formalize $T(n)$ for the best case and the worst case as follows:

Definition 6: *Best Case Running Time for Insertion Sort*

```
⊢ ∀ n c1 c2 c3 c4 c7.  i_sort_bc_t n c1 c2 c3 c4 c7 =
                                  c1 n + (c2 + c3 + c4 + c7)(n - 1)
```

Definition 7: *Worst Case Running Time for Insertion Sort*

```
⊢ ∀ n c1 c2 c3 c4 c5 c6 c7.  i_sort_wc_t n c1 c2 c3 c4 c5 c6 c7 =
    c1 n + (c2 + c3 + c7)(n - 1) +
    c4 sum(2, n - 1)(λj.  j) + c5 sum(2, n - 1)(λj.  j - 1) +
                                        c6 sum(2, n - 1)(λj.  j - 1)
```

where `sum` represents the summation of a real sequence function in HOL, i.e.,`sum(m, n)f` = $\sum_{i=m}^{m+n-1} f$. Now, the next step is the formal verification of the bounds of the running time of the insertion sort algorithm. We formalize the bounds of $T(n)$ using the formal definition of O notation (Definition 1) formalized in Chapter 1 as the following theorem.

Theorem: 6 *Asymptotic Bounds of Running Time of Insertion Sort*

```
⊢ ∀ n c1 c2 c3 c4 c5 c6 c7.
   0 < c1 ∧ 0 < c2 ∧ 0 < c3 ∧ 0 < c4 ∧ 0 < c5 ∧ 0 < c6 ∧ 0 < c7
     ⟹ ( (λn.  i_sort_bc_t n c1 c2 c3 c4 c7 ) ∈ (BigO n) ∧
       (λn.  i_sort_wc_t n c1 c2 c3 c4 c5 c6 c7 ) ∈ (BigO n pow 2))
```

The proof of Theorem 6 is based on the Definitions 1, 6 and 7 and some simple arithmetic and set theoretic properties. The main purpose of including this first case study is to illustrate the process of analyzing the asymptotic bounds of a given algorithm. We presented the analysis of the O bound of the Insertion sort algorithm and similarly any other asymptotic bound can be formally analyzed for any other algorithm.

The proof of Theorem 6 is based on the Definitions 1, 6 and 7 and some simple arithmetic and set theoretic properties. The main purpose of including this first case study is to illustrate the process of analyzing the asymptotic bounds of a given algorithm. We presented the analysis of the O bound of the Insertion sort algorithm and similarly any other asymptotic bound can be formally analyzed for any other algorithm.

4.2 Uniqueness of Array Elements

As our second case study, consider an algorithm that is used to determine whether the given array is composed of all non-repetitive elements or, in other words, all the elements of the given array are unique. This algorithm is composed of two parts. First, we sort the array using the insertion sort algorithm, described in the previous section. Secondly, we scan the sorted array to check its consecutive elements for equality by using the linear search algorithm [30].

Now, we will present the formal verification of the asymptotic bounds of the running time of this algorithm. As illustrated in the case of the insertion sort algorithm, the first step is to formalize the running time of the given algorithm, which can be done based on the following definitions.

Definition 8: *Running Time of Linear Search*

```
⊢ ∀ n c.  search n c = c n
```

Definition 9: *Worst Case Running Time of Uniqueness of Array Elements*

```
⊢ ∀ n c1 c2 c3 c4 c5 c6 c7.  t_unique n c1 c2 c3 c4 c5 c6 c7 =
      (i_sort_wc_t n c1 c2 c3 c4 c5 c6 c7) + (search n c))
```

Now we will formally verify the bounds of the running time of this algorithm.

Theorem: 7 *Asymptotic Bounds of Uniqueness of Array Elements*

```
⊢ ∀ n c1 c2 c3 c4 c5 c6 c7.
0 < c1 ∧ 0 < c2 ∧ 0 < c3 ∧ 0 < c4 ∧ 0 < c5 ∧ 0 < c6 ∧ 0 < c7
     ⟹ (λn.  t_unique n c1 c2 c3 c4 c5 c6 c7 ) ∈ (BigO (n pow 2)
```

The proof of Theorem 7 is based on the Definitions 1, 7, 8 and 9 and Theorems 2 and 6 along with some simple arithmetic and set theoretic properties. This example illustrates the usefulness of one of our formally verified properties (Theorem 2) in the asymptotic analysis of algorithms. It was due to the availability of this formally verified result that

the formal reasoning required for Theorem 7 was very straightforward and required merely a few lines of HOL code.

Chapter 5

Conclusions and Future Work

In this thesis, we presented a higher-order-logic formalization of Asymptotic notations. First of all, we formalized the definitions of O, Θ, Ω, o and ω. Then by using these definitions, we formally verified their properties such as transitivity, symmetry, transpose symmetry, reflexivity, sum, product, min, max etc. using the HOL4 theorem prover. The reported formalization facilitates the process of formally analyzing the complexities of algorithms using these notations in HOL4. To demonstrate the practical usefulness of our work, we presented two case studies for the formal analysis of the insertion sort algorithm and a unique array identification algorithm. To the best of our knowledge, this kind of formal analysis involving asymptotic notations has not been reported in the open literature so far.

In real-world applications, most of the algorithms involve more than one parameters which influence overall performance of these algorithms. For example, running time of a graph algorithm depends on both the number of vertices and number of edges. In order to formally analyze such applications, our formalization can be extended to formalize alternative definitions of asymptotic notations involving multiple variables [23, 10]. Another interesting application domain of our formalization is cryptography [33] where asymptotic notations are used to estimate the size of the key so that it will be infeasible to break a system using given number of steps. Similarly, asymptotic notations are frequently used in security assessment of authentication protocols, such as the security proof of password authentication protocols [6] and the reported formalization can play a vital role in this kind of security-critical analysis.

References

[1] The HOL System Description, 2011.

[2] B. Akbarpour and L. C. Paulson. Metitarski: An Automatic Prover for the Elementary Functions. In *AISC/MKM/Calculemus*, pages 217–231, 2008.

[3] O. L. Astrachan and D. W. Loveland. The use of lemmas in the model elimination procedure. *J. Autom. Reason.*, 19(1):117–141, August 1997.

[4] J. Avigad and K. Donnelly. Formalizing O Notation in Isabelle/HOL. In *Automated Reasoning*, volume 3097 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2004.

[5] P. G. H. Bachmann. *Analytische Zahlentheorie, Bd 2: Die Analysische Zahlentheorie.* Teubner, Leipzig, Germany, 1894.

[6] H. Bakuri. Security Analysis and Implementation of Password-based Cryptosystem. Master's thesis, ECE, School of Engineering and Design Brunel University, 2005.

[7] B. Beckert and Joachim Posegga. leantap: Lean tableau-based theorem proving (extended abstract). In *PROC. CADE-12, LNAI 814*, pages 793–797. Springer Verlag, 1994.

[8] Yves Bertot. A short presentation of coq. In *TPHOLs*, pages 12–16, 2008.

[9] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[10] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms.* McGraw-Hill, 2nd edition, 2001.

[11] Wendell P Cropper Jr. and Patti J Anderson. Population dynamics of a tropical palm: use of a genetic algorithm for inverse parameter estimation. *Ecological Modelling*, 177(1-2):119–127, 2004.

[12] Andrea Gasparri, Stefano Panzieri, and Federica Pascucci. A spatially structured genetic algorithm for multi-robot localization. *Intelligent Service Robotics*, 2(1):31–40, 2008.

[13] Gilbert and Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(3):345 – 358, 1992.

[14] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge Press, 1993.

[15] C. Gutwenger and P. Mutzel. A Linear Time Implementation of SPQR-Trees. In *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2001.

[16] Robert M Haralick and Linda G Shapiro. *Computer and Robot Vision*, volume 1. Addison-Wesley, 1992.

[17] J. Harrison. Formalized Mathematics. Technical Report 36, Turku Centre for Computer Science, 1996.

[18] J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[19] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[20] J. Harrison. A List of Theorem Provers, 2011.

[21] John Harrison. Hol light: A tutorial introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD96),*

volume 1166 of Lecture Notes in Computer Science, pages 265–269. Springer-Verlag, 1996.

[22] J. E. Hopcroft and R. E. Tarjan. Dividing a Graph into Triconnected Components. *SIAM Journal of Computing*, 2(3):135–158, 1973.

[23] R. R. Howell. On Asymptotic Notation with Multiple Variables. Technical report, Dept . of Computing and Information Sciences Kansas State University, Manhattan, KS 66506 USA, January 2008.

[24] N. Iqbal. Formalization of Asymptotic Notations in HOL4 - Proof Script, 2012.

[25] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[26] Matt Kaufmann and J. Strother Moore. An acl2 tutorial. In *TPHOLs*, pages 17–21, 2008.

[27] S. A. Khuri and S. Xie. On the Numerical Verification of The Asymptotic Expansion of Duffing's Equation. *International Journal of Computer Math.*, 72(3):325–330, 1998.

[28] D. E. Knuth. *Big Omicron and Big Omega and Big Theta*. ACM SIGACT News, 8:1824, 1976 Teubner, Leipzig, Germany, 1909.

[29] E. Landau. *Handbuch Der Lehre Von Der Verteilung Der Primzahlen*. Teubner, Leipzig, Germany, 1909.

[30] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[31] Jun Liu, Luis Martinez Lopez, Yang Xu, and Zhirui Lu. Automated reasoning algorithm for linguistic valued lukasiewicz propositional logic. In *Proceedings of the 37th International Symposium on Multiple-Valued Logic*, ISMVL '07, pages 29–, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Anderson Gonalves Marco, Alexandre Souto Martinez, and Odemir Martinez Bruno. Fast, parallel and secure cryptography algorithm using lorenz's attractor. *International Journal of Modern Physics C*, 21(03):365, 2010.

[33] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[34] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[35] Mark-Jan Nederhof and Giorgio Satta. Algorithms and theory of computation handbook. chapter Algorithmic aspects of natural language processing, pages 23–23. Chapman & Hall/CRC, 2010.

[36] D. Nilsson. An efficient algorithm for finding the m most probable configurations in probabilistic expert systems. *Statistics and Computing*, 8(2):159–173, June 1998.

[37] Sam Owre and Natarajan Shankar. A brief overview of pvs. In *TPHOLs*, pages 22–27, 2008.

[38] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[39] Rolf and Haenni. Using probabilistic argumentation for key validation in public-key cryptography. *International Journal of Approximate Reasoning*, 38(3):355 – 376, 2005. ¡ce:title¿Selected papers from ECSQARU-2003¡/ce:title¿.

[40] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 4th edition, 2003.

[41] J. Schumann. Sicotheo - simple competitive parallel theorem provers based on setheo. Technical report, In Parallel Processing for Artificial Intelligence 3, Machine Intelligence and Pattern Recognition, 1995.

[42] Konrad Slind and Michael Norrish. A brief overview of hol4. In *TPHOLs*, pages 28–32, 2008.

[43] Tanel Tammet. Gandalf, 1997.

[44] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The isabelle framework. In *TPHOLs*, pages 33–38, 2008.